Temporal Data Management and Incremental Data Recomputation with Wide-column Stores and MapReduce

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

M. Sc. Yong Hu

Datum der wissenschaftlichen Aussprache: 31.07.2017

Dekan: Prof. Dr.-Ing. Stefan Deßloch

Berichterstatter: Prof. Dr.-Ing. Stefan Deßloch

Berichterstatter: Prof. Dr.-Ing. Norbert Ritter

D 386

For My Family

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr.-Ing. Stefan Deßloch, for his continuous support, for allowing me to manage my time flexibly, and, above all, for offering me the opportunity of doing research under his guidance. Without his help and advice, this work could not be done. Furthermore, I would like to thank Prof. Dr.-Ing. Norbert Ritter for taking the role of the second examiner and Prof. Dr.-Ing. Jens Schmitt for acting as chair of the PhD committee. Their suggestions and feedback greatly contributed to this dissertation.

I would like to thank all my colleagues for giving me an excellent time in TU Kaiserslautern. Many thanks go to Dr. Thomas Jörg, Johannes Schildgen and Weiping Qu for kind help and research suggestions during my PhD study; to Caetano Sauer for excellent tourism in Paris; and to Heike Neu and Steffen Reithermann for administrative support. The irregular tee and coffee break with Dr. Daniel Schall and regular lunch together with Dr. Yi Ou, Dr. Boris Stumm, Joachim Klein, Xiaofeng Xia, Huiying Duan, Hongzhe Jia, Evica Milchevski, Koninika Pal, Kiril Panev and Prof. Dr.-Ing. Sebastian Michel have been relaxing and interesting.

Most importantly, I am very grateful to my parents, Nianhua Jiang and Yueming Hu for their love, patience, support and encouragement. Special thanks go to my wife Rebecca and my three daughters, Lea, Anna and Ida who have been with me through all the good and hard time.

Yong Hu

Kaiserslautern, September 2017

Abstract

In recent years, "Big Data" has become an important topic in academia and industry. To handle the challenges and problems caused by *Big Data*, new types of data storage systems called "NoSQL stores" (means "Not-only-SQL") have emerged.

"Wide-column stores" are one kind of NoSQL stores. Compared to relational database systems, wide-column stores introduce a new data model, new IRUD (Insert, Retrieve, Update and Delete) semantics with support for schema-flexibility, single-row transactions and data expiration constraints. Moreover, each column stores multiple data versions with associated timestamps. Well-known examples are Google's "Big-table" and its open sourced counterpart "HBase". Recently, such systems are increasingly used in business intelligence and data warehouse environments to provide decision support, controlling and revision capabilities.

Besides managing the current values, data warehouses also require management and processing of historical, time-related data. Data warehouses frequently employ techniques for processing changes in various data sources and incrementally applying such changes to the warehouse to keep it up-to-date. Although both incremental data warehousing maintenance and temporal data management have been the subject of intensive research in the relational database and finally commercial database products have picked up the ability for temporal data processing and management, such capabilities have not been explored systematically for today's wide-column stores.

This thesis helps to address the shortcomings mentioned above. It carefully analyzes the properties of wide-column stores and the applicability of mechanisms for temporal data management and incremental data warehouse maintenance known from relational databases, extends well-known approaches and develops new capabilities for providing equivalent support in wide-column stores.

Contents

Chapter 1

Introduction

"Big Data" currently receives significant attention in research and industry [ML09, Uni, DMGG15, HWCL14], and it is generally characterized as "4v"s [Jac09, Fvs], namely, volume, velocity, variety and veracity. To deal with the issues and challenges due to Big Data, new types of data storage systems called "NoSQL stores" [Nosa, SF12, RW12] have emerged. These systems manage tremendous data volume in off-the-shelf commodity hardware with support for elastic scalability, high availability and robust fault tolerance.

"Wide-column stores" are one type of NoSQL stores which organize the data in a structured way and store the data which belong to the same *column* contiguously on disk. Such systems can be conceived as key-value stores with support for a timestamped/versioned data model, new IRUD (Insert, Retrieve, Update and Delete) operations, single row transactions and value or data expiration constraints. Moreover, in contrast to relational database systems, data analysis tasks performed against wide-column stores are usually implemented by programmers at the application layer as wide-column stores lack a powerful query processing engine. Well-known examples are Google's "Big Table" [CDea06, CDea08] and its open-sourced counterpart "HBase" [Hba, Geo11].

1.1 Motivation

Recently, wide-column stores are increasingly utilized as data sinks, e.g. business intelligence tools and data warehouse environments to provide decision support, controlling and revision capabilities. A data warehouse is a data

repository which stores extracted data values from various data sources for further data analysis. Besides managing up-to-date values, a data warehouse also requires to store and process historical and time-related data.

Each table in a data warehouse [KC04] can be viewed as a materialized view which is derived from different data sources. For processing changes made on data sources, a data warehouse incrementally applies such changes to the corresponding tables to keep them up-to-date. Although both incremental data warehouse maintenance [LGM96a, LYea00, MQea97, JMS09] and temporal data management [CC87, DBS96, SAea94, GR09, MZ15] have been the subject of research in the relational database area, such capabilities have not been systematically explored for today's wide-column stores. For example, in contrast to relational database systems, each column in a wide-column store can maintain multiple data versions. In consequence, the *tuple-time-stamping model* [JSS94, JSS, BOea98] (in which each a tuple is associated with a temporal interval) is not the only choice for modeling temporal data. More discussions of suitable temporal data models for wide-column stores will be given in Chapter 4.

1.2 Goals

To systematically explore the capabilities of temporal data management and temporal data processing for wide-column stores, this thesis mainly focuses on the following two aspects:

1. How to effectively manage and process temporal data in wide-column stores.
2. How to incrementally re-compute materialized temporal computation results in wide-column stores.

1.3 Contributions

The thesis carefully analyzes the properties of wide-column stores and studies the applicability of mechanisms for temporal data modeling and processing and incremental data warehouse maintenance known from relational databases. Based on the characteristics of wide-column stores, we extend well-known approaches and develop new capabilities to provide equivalent support:

- We propose two alternative temporal data representations [HD14a, HD15b], namely, explicit-history representation (EHR) and tuple-time-stamping representation (TTR) based on the data model supported by wide-column stores. The new data models avoid wrong or misleading temporal query results caused by the default temporal interval representation.

- To analyze data in wide-column stores, users can either write low-level programs, such as MapReduce [Dea04] procedures or utilize high-level languages, such as Pig Latin [Pig] or Hive [Hiv]. Nevertheless, all of these approaches require users to explicitly implement the desired semantics of temporal query processing. Hence, two temporal operator models (based on the previous two temporal table representations), i.e. *TTRO* and *CTO* [HD14a, HD15b] are defined to facilitate the temporal query specification.

- To incrementally update the data warehouse, we first describe how to model and extract temporal deltas (change data). We build a logical delta model to support the characteristics of wide-column stores, such as multiple granularities of operations, schema flexibility and expiration of data items. Based on the temporal data representations and the corresponding temporal operator models, two temporal delta representations are described, namely, row-level representation and (enhanced) attribute-level representation. To capture temporal deltas in required formats, we investigate different change-data-capture (CDC) [HQ12, HD13, HD14b] approaches, based on Timestamps, Audit-columns, Triggers and Snapshot-differential. The Time-stamped-based approach developed in this thesis is a new, wide-column-store-specific approach. In view of the characteristics of each individual CDC approach, not all of them can produce complete delta sets.

- For incrementally propagating complete temporal deltas, we first introduce a set of temporal propagation rules based on the TTRO model. When the CTO model is used, the column-level temporal delta propagation can lead to additional tuple reconstruction if the complete tuples are needed. However, such reconstruction tasks can be avoided by an enhanced attribute-level-representation [HDH16]. Moreover, we also address the issues where, due to the capture mechanisms in use, deltas are only partially available [HD15a].

1.4 Outline

In this thesis, we present our work on temporal data management and incremental data re-computation with wide-column stores and MapReduce. The thesis is mainly divided into 4 parts.

1. Introduction and Preliminaries

    - Chapter 2 first gives a short overview of NoSQL stores and describes the characteristics of wide-column stores. As wide-column stores usually do not offer the sophisticated SQL engines, we then describe the popular parallel data processing framework for wide-column stores.

2. Temporal data management and processing in wide-column stores

    - In Chapter 3, we review the general approaches for temporal data modeling and temporal data processing. The review covers both research efforts in the last decades and state-of-the-art methods utilized by commercial database vendors.
    - Chapter 4 describes how to manage temporal data in wide-column stores. We first analyze the semantics of timestamp (TS) supported by wide-column stores and indicate the drawbacks of its default temporal interval representation. Then, two explicit temporal interval representations are proposed.
    - We represent how to process temporal data in wide-column stores in Chapter 5. Two temporal operator models (corresponding to the previous two temporal models) are defined to facilitate the temporal query specification.

3. Incremental temporal data processing

    - We review the existing incremental re-computation approaches proposed in both non-temporal and temporal context in Chapter 6.
    - Chapter 7 describes the issues of temporal change-data capture (CDC). We first introduce our temporal delta model and its corresponding delta representations. Then, five feasible CDC approaches are depicted.

- The incremental temporal data re-computation based on two temporal operator models are described in Chapter 8. This chapter only considers how to propagate the complete temporal deltas.
- The incomplete (partial) temporal delta propagation is introduced in Chapter 9.

4. Conclusion

- In Chapter 10, the thesis ends with a conclusion drawn from the preceding studies and provides an outlook for future work.

Chapter 2

NoSQL data stores

In this section, we first give an overview of NoSQL [Nosa, Ria, Red, Ber, CDea06, Hba, Mon, Cou, Cas, EFH+11] history and briefly describe various NoSQL data stores. Then, we introduce wide-column stores (WCS) with respect to their data model, data operation model and data consistency model. As WCS usually lacks the high-level query languages and the sophisticated query engines, we finally describe the popular parallel data processing frameworks utilized by WCS.

2.1 Overview

Although *NoSQL* is a well-known term in these days, there still exists a lot of debates and discussions about it. In this section, we first describe the history of "NoSQL" to clarify where it comes from and what it means. Then, we introduce various types of NoSQL stores and indicate their general characteristics.

2.1.1 NoSQL history

Relational database systems have dominated the database area for several decades. They are always considered as the first choice to build applications, from small to large. However, at the beginning of 2000, their dominance has been challenged by a new type of database system called *NoSQL stores* [Nosa, Ria, Red, Ber, Hba, CDea06, Mon, Cou, Cas, EFH+11, SF12]. The emergence of NoSQL stores is motivated by the rapidly growing usage of

Internet services such as social network, search engine, blog, e-commerce and so on. These web-based applications bring more critical system requirements for database systems, such as high throughput, elastic scalability, flexible data structure, 24/7 availability and robust fault-tolerance. Moreover, people wish to obtain the previously described features at a low-cost (or even free).

The term *NoSQL* first appeared in 1998. It was used by Carlo Strozzi [CSN] to name his database. His database stores tables as plain text files without supporting the SQL query language. In 2009, Eric Evans, a software developer at Rackspace, reused this term during a conversation [NoSb] with an organizer who wanted to organize a meeting for *open-sourced, distributed, non-relational databases*.

After that, the term "NoSQL" was widely used. As no one ever gave an explicit definition of "NoSQL stores", different people have different understanding. At the beginning, it was obvious that NoSQL stores do not support SQL. For example, if we trace back to the early versions of NoSQL data stores, such as HBase [Hba], Cassandra [Cas] and MongoDB [Mon], we find they did not have SQL support. This feature coincides with some person's understanding "No SQL".

However, after a while, people found that querying the data in NoSQL stores was cumbersome and complicated (as most NoSQL stores require users to implement query processing by themselves). In consequence, people decided to rebuild high level languages for NoSQL stores to facilitate the query specification. In these days, people prefer to interpreting "NoSQL" as "Not only SQL" which means NoSQL stores can be seen as a complement of relational database systems and offer more choices when building desired applications.

Please notice that the term "NoSQL stores" used today mainly denotes database systems developed with the emergence of web 2.0 (in early 21st century). Non-relational database systems which were developed before that time are usually not considered as "NoSQL stores", e.g. object-oriented database systems [RW12].

2.1.2 Classification of NoSQL stores

Today, there are numerous NoSQL stores in both academia and industry. Generally, they can be classified using four categories, which are *key-value stores*, *wide-column stores*, *document-oriented stores* and *graph stores*.

Key-value stores [Ria, Red, Ber] are the simplest NoSQL stores which organize and record data as key-value pairs. They utilize a *hash-table* data structure in which each tuple is allocated, distributed and retrieved based on the key. In general, a key for each tuple can be automatically generated by database system or provided by user. The structure of a value associated with a key may vary. For example, the value can have a simple scalar type such as *string* or *integer* or a complex type such as *list* or *map*. As key-value stores are schema-less, there is no system-level meta-data which can be exploited to interpret the data. Moreover, the values of key-value stores are usually stored as byte-arrays. In consequence, we can store two totally different key-value pairs in the same key-value store, e.g. the value part of tuple A has a relational structure where the value part of tuple B utilizes a *JSON* data representation. These characteristics point to the fact that detecting and extracting the real structure of a value is the responsibility of the application program.

As the structure of values is uncertain, key-value stores generally do not support high-level query languages. Users can acquire fast access through simple CRUD (create, retrieve, update and delete) operations, but more complex query processing, such as *join* or *group-by* has to be implemented manually. Well-known examples are Riak [Ria], Redis [Red] and Berkeley DB [Ber].

In 2006, Google published a paper [CDea06] to describe how the database in their back-end works. This paper can be considered as the first paper that provides a detailed description of *wide-column stores* (WCSs). Like relational database systems, WCSs keep the notion of tables, rows and columns. Moreover, they introduce a new concept called "column family". A column-family can be seen as a prefix of column names which denotes all the columns which belong to the same column-family will be stored contiguously on disk. Although WCSs have relational structure, no "*NULL*" values are stored. This property is usually known as "*sparse data sets*". When creating a table, only table name and column-family name must be predefined. The name of column is only specified when data is inserted. Moreover, each column can contain multiple data versions which are sorted based on their corresponding timestamps. Unlike relational database systems, the DML (data manipulation languages) of WCSs supports various data granularities. For example, the delete operations can be applied to a *data version*, a *column* and a *column family*. Well-known examples of wide-column stores are *Big table*, *HBase* and *Cassandra* [Cas].

As there are usually no built-in SQL engines for WCSs, users have to write low-level programs to analyze the data. Nevertheless, large IT companies such as Yahoo! and Facebook attempt to build high-level languages for query processing. More details will be given in Section 2.3.

Document-oriented stores can be seen as a refinement of key-value stores. Each table in a document-oriented store is called *collection*. Each collection is composed by a set of documents (analogy to tuples). Each document has an unique ID and a series of attributes (which can be genreated on-the-fly). Internally, document-oriented stores manage data by using BSON (binary JSON) format. Different from key-value stores and wide-column stores, document-oriented stores support for secondary indices, aggregation and views. Moreover, users can access data in document-oriented stores by exploiting high-level query languages and user-defined functions. Well-known examples are *MongoDB* [Mon] and *CouchDB* [Cou].

Graph stores [RWE11] are widely used in social network applications and are based on graph theory. In contrast to the relational data model, data in graph stores are organized as *nodes* and *edges*. Nodes represent entities, e.g. people or accounts. Edges denote the relationships between nodes. Generally, each node and each edge can contain any number of properties or attributes which specify their individual information. Again, each property is formed as key-value pair. In general, each node in graph stores can have different meta-data. Moreover, the attributes of each node or edge can be indicated on-the-fly. To traverse the data in a graph database, either *depth-first search* or *breadth-first search* can be exploited based on the selection criteria. To query the data in graph stores, users can write low-level programs or use third-party tools.

Although the various NoSQL stores have different properties, all of these systems share a number of common design features:

- *Horizontal scalability*: Databases must support a horizontal scale-out strategy using many commodity hardwares.
- *Network-parition tolerance*: Although the network in the cluster is not stable, messages transferred and received between nodes must be guaranteed.
- *Basic Availability*: Systems must always respond to the user's requirements even when a failure occurs, such as a network failure or power outage.

- *Elasticity*: Computation nodes in the cluster can be added or removed dynamically without heavily influencing the usage of database systems.
- *Horizontal data partitioning*: Data content is physically partitioned and distributed.
- *Multiple data replica*: For recovery and performance reasons, one data object can have multiple data replicas which reside in different physical data nodes.
- *De-normalized tables*. One column can maintain multiple data versions (which violates first-normal-form). As NoSQL databases do not support join operation, different but logically related tables are stored as a single physical table.
- *Schema flexibility*: Data can be inserted into the database without predefining the schema.
- *Simple access pattern*: The database systems lack sophisticated query processing and only support a simple client API. Complex query tasks have to be implemented at the application layer.
- *Fault tolerance*: Databases have the ability to handle certain failures automatically without requiring the concern of developers or system administrators.
- *High throughput*: The database systems must have the capability to deal with large amounts of data in a reasonable amount of time.
- *Weak consistency*: In order to guarantee availability during network failures, NoSQL stores usually support weak consistency.

2.2 Wide-column stores

In this section, we will give a detailed description of wide-column stores (WCSs) [Geo11, Hew10, SF12]. We first introduce the data model of WCS and indicate how its data is organized and stored in a distributed file system. Then, we use examples to explain how users can modify the state of WCSs. Finally, we introduce their data consistency models and compare WCSs with relational database systems.

2.2.1 Data model

In addition to the concepts of table, row and column, WCS introduces a new concept called “column family”. In analogy with RDBMS, a column-family can be considered as a ”table” and columns which belong to the same column family will be stored contiguously on disk. Each table in a WCS is partitioned based on the row keys. For a given tuple, one column may store multiple data versions that are sorted by the corresponding timestamps (TSs). The value of TS should be unique and indicates when the data version is generated. Each data version in a column is stored as a key-value pair in a distributed file system. The key part is composed of information such as row key, column-family name, column name and TS. The value part contains the data value. Moreover, users can indicate a ttl (time-to-live) property for each column family to denote the life time of a data item. When data items expire, WCSs will make them invisible. The actual data deletion occurs at data compaction time.

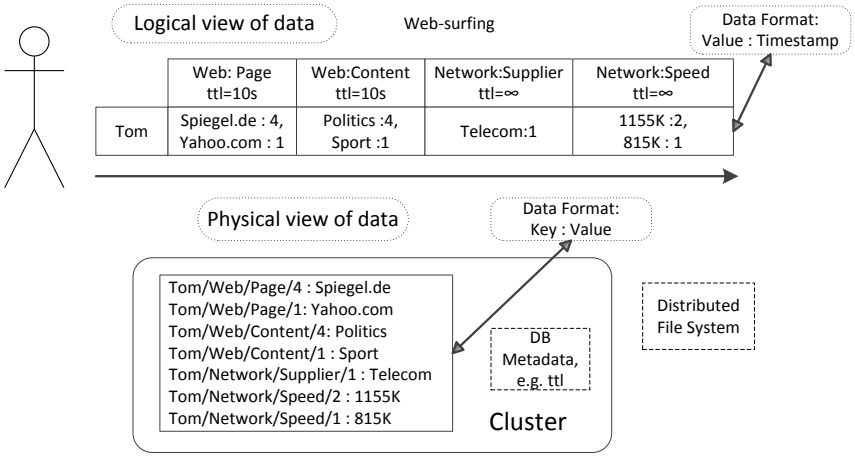


Figure 2.1: Example of wide-column stores

Figure 2.1 shows an example defined in ”HBase” to illustrate the aforementioned data model concepts. The ”Web-Surfing” table records the information when a user browses the internet. It contains two column families, i.e. ”*Web*” and ”*Network*” and each column-family includes 2 columns. For row ”Tom”, ”Web:Page” column contains two data versions with the corresponding timestamps. The ttl property for the ”Web:Page” column is 10

seconds, which indicates data version "Yahoo.com:1" will be invisible at 11 seconds. For the "Network:Supplier", the value of ttl is infinite which denotes the data versions in that column have no time expiration constraints. The logical view of data represents the data structure as presented to the user. In contrast, the key-value pairs which are persisted in distributed file system depict the physical view of data.

2.2.2 Data processing model

In an RDBMS, insertion and deletion will generate and discard a complete row. An update of a row will overwrite the old value by a new value. However, WCSs do not distinguish update from insertion. A new data version (no matter the row key already exists in the table or not) for a specific column will be generated by the "Put" command. [1] When issuing a "Put" command, users need to denote the parameters such as row key, column-family name, column name, value and TS (optional). For example, to generate the data version "1155K:2" for "Network:Speed" column for row "Tom" in Figure 2.1, a Put command will be first initialized with the row key "Tom", and then the $Put.add(Network, Speed, 1155K)$ method is utilized to add the desired column value. After executing the "Put" command, a new data version will be inserted without overwriting the older ones.

Following the data model of WSC, data deletions can partially prune a row by eliminating the column values at various granularities, namely, *data version*, *column* and *column family*. A row deletion is generally realized by a set of column-family deletions. When issuing a "Delete" command, users need to specify parameters such as the row key. For the different delete types, more information such as column-family name, column name and TS can be indicated. For example, to delete the data version "1155K:2" in the "Network:Speed" column for row "Tom" in Figure 2.1, a delete object is initialized with the row key "Tom" and the $Delete.deleteColumn(Network, Speed, 2s)$ method is exploited to indicate the deleted data version. In contrast to RDBMS, delete operations in the WCS will not delete data right away but insert a marker called "*tombstone*" to mask the data values whose TSs are less than or equal to the TS of the tombstone. The physical data deletions take place at data compaction time.

[1] Note that "Put" is used by HBase and Cassandra uses "Set", in the rest of the thesis we use "Put" to represent both.

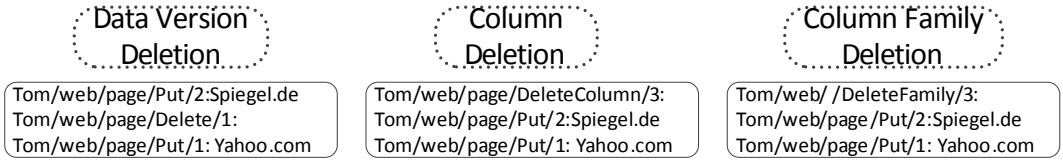


Figure 2.2: Various deletion levels of wide-column stores

Figure 2.2 shows how a tombstone is formatted as key-value pair with respect to each individual deletion granularity in HBase. For each tombstone, the type of deletion is embodied into the key part, such as *Delete*, *DeleteColumn* and *DeleteFamily*, where the value part is empty.

To read the content of a WCS table, users can either use a "Get" command to read the data versions of a specific row or a "Scan" method to access the whole table. For example, for retrieving the contents of column "Web:Page" for row "Tom", a Get command will be first initialized with the row key "Tom", and then we can call $Get.addColumn(Web, Page)$ method to obtain the desired contents. We summarize the operations supported by WCSs in table 2.1:

Data commands	**Descriptions**
Put	Generate new data versions
Delete	Delete data versions
Get	Obtain the contents of a single row
Scan	Access the whole table contents

Table 2.1: Commands supported by wide-column stores

2.2.3 Data consistency model

The "*CAP*" theorem [GL02, Bre12] which was proposed by Eric Brewer in 2000 has become a guideline for designing NoSQL systems. CAP is the shorthand for *consistency*, *availability* and *partition tolerance*. The premises of the CAP theorem are:

1. One data object may have multiple data replicas distributed across different physical nodes.

2. A system (cluster) can't guarantee message delivery (message can get lost).

3. Each node can execute read and write operations, and propagate new values to the other nodes.

The first premise implies that at a certain point of time, if the system cannot guarantee strong consistency, the values of data replicas may be different; the second precondition means the system doesn't guarantee the communication between each node through the network; the third one implies each node in the cluster is equal and any of them can be a coordinator node which is responsible for managing and scheduling processing tasks or a worker node which is responsible for executing processing tasks.

The meaning of each CAP property is described as followed:

- Consistency (C): All the replicas of the same data object share the same value at the same time.

- Availability (A): The system can accomplish the data access (read and write) requests during an acceptable response latency.

- Partition tolerance (P): The system has the ability to handle the "message lost" issue and continues to operate.

The theorem states that any highly scalable system can only satisfy two out of three of them. Let's take a look at the combinations of these properties and indicate what features the corresponding system will get:

- $A+C$: implies there are no data replicas existing in different nodes (all replicas reside on one data node).

- $P+C$: if the system wants to keep consistency, each replica must hold the same data value. Data replica synchronization will influence the availability. For example, if a write operation is issued, the client has to wait until data synchronization completes. However, our premise 2 makes this waiting time undecidable with respect to the network partition. It will cause high latency and give the client the impression that the system is down.

- $A + P$: provides low latency, but cannot guarantee data consistency. This implies that a client can see different versions for the same data object and requires the applications to handle the stale and inconsistent data.

To build a data storage system in a large cluster, there is a general agreement that the partition tolerance (P) is an important factor. Hence, besides satisfying P, different NoSQL stores choose to satisfy the rest of CAP properties due to the system and application requirements. In the WCS context, HBase prefers $C + P$ where Cassandra fulfills $A + P$ with tunable consistency [Com08, Hew10].

To achieve data consistency, HBase routes all the data read and write requests for the same row to the same data node (called "*RegionServer*" in HBase). This strategy guarantees that different applications will not see various data replicas at the same time. The replicas in HBase are mainly utilized for crash recovery rather than the availability or the performance enhancement. However, in contrast to RDBMSs, HBase only supports single-row transactions.

Cassandra supports tunable data consistency by requiring users to explicitly set the relationships between the number of replicas (N), the number of replicas needed to acknowledge receipt of updates (W) and the number of replicas contacted when data is read (R). For example, when *strong consistency* is required, formula $R+W > N$ should hold. Generally, the throughput of read and write operations of Cassandra is higher than HBase [CST+10], as it supports eventual consistency, i.e. $W < N$ and $W + R <= N$. Although weak consistency can reduce the response time for data accessing, it requires applications to deal with the inconsistent data and hence increases the burden of programmers.

2.2.4 Comparison with relational database systems

In this section, we summarize the differences between RDBMS and WCS in table 2.2. In contrast to the previous sections, we add some nontechnical aspects, e.g. hardware requirements, cost and so on.

Aspects	RDBMS	WCS
Hardware	powerful machines	commodity hardware
Logical Data model	relation	relation (column family)
Row key	optional	mandatory
Data versioning	not supported	supported
Schema flexibility	no	yes
Normalized data	yes	no
Physical data representation	record	key-value pairs
Operational granularity	row level	various levels
SQL support	yes	no
Secondary index	supported	not supported
View	supported	not supported
Transaction	yes	limited (single-rows)
Open-sourced	yes	yes
Cost	vendor-specific	free

Table 2.2: Comparisons between RDBMS and WCS

2.3 Data processing framework for wide column stores - state of the art

As WCSs usually lack sophisticated SQL engines, we give a short overview of the state-of-the-art for parallel data processing frameworks which are mostly utilized for WCSs. We first introduce the MapReduce [Dea04, Whi12] framework which was proposed by Google in 2004. Then, we describe two high-level query languages, namely *Hive* [TSea09, TSea10, LLea, Hiv] and *Pig Latin* [ORea, Pig], which are built on top of MapReduce.

2.3.1 MapReduce framework

MapReduce [Dea04, Whi12] is a programming model and an associated implementation designed for parallel processing of massive data sets in clusters that are built using commodity hardware. Its programming model, namely *Map* and *Reduce*, is inspired by functional programming. The tasks such as data partitioning, intermediate results caching and delivery, distributed task

scheduling, network communication and fault tolerance are handled by the MapReduce run-time system.

The programming model of MapReduce shown in the original paper [Dea04] is described as follows:

$$map : (k_1, v_1) \rightarrow list(k_2, v_2) \tag{2.1}$$

$$reduce : list(k_2, v_2) \rightarrow list(v_2) \tag{2.2}$$

- Map function. A Map function takes an input key-value pair (k_1, v_1) and produces a set of intermediate key-value pairs (k_2, v_2). The $list(k_2, v_2)$ can contain several elements which share the same key.

- Reduce function. A Reduce function merges the intermediate key-value pairs $(k_2, list(v_2))$ together to produce a smaller set of values $list(v_2)$.

The definition of Reduce function from the original MapReduce paper suggests that the output key of the Reduce function must be the same as the input key. However, in real applications, the output key of the Reduce function is always different from the input key. Hence, we use the following definition for the Reduce function:

$$reduce : list(k_2, v_2) \rightarrow (k_3, v_3) \tag{2.3}$$

Please note that v_3 could be either a single value or a set of values, e.g. list.

Algorithm 1 shows an example which uses the MapReduce paradigm to count the number of users for each Internet supplier based on Figure 2.1. The Map function takes the value of each row as input and produces a pair of $(Internet - supplier, 1)$ for each present Internet-supplier. To reduce the amount of data delivered to the Reduce nodes, we insert a *Combine* function between Map and Reduce functions. The functionality of Combine is the same as the Reduce function which summarizes the count for each Internet supplier. However, different from the Reduce function, Combine only aggregates the values, which exist on the same physical (worker) node. After Combine is finished, the MapReduce framework will deliver the intermediate key-value pairs to different reducers. In general, this process is accomplished by the *Partitioner*. The default partition strategy supported by MapReduce is *hash partitioning*, namely $fun(key, value)\%n$, in which we first calculate a value using function $fun(key, value)$ (by the way, only utilizing key or value

for calculation is also possible), and then a *mod* (%) operation is executed based on the number of reducers (indicated by n). After data partitioning finishes, the data will be sent to each Reducer for final calculation. Note that, all Reducers have to wait until all Mappers (or Combiners) accomplish their work and the intermediate values are persistent on disk. This strategy guarantees that

- if something goes wrong in the Map function, no intermediate results will be seen by Reducers, and
- if errors occur during the execution of the Reduce function, it is always possible for Reducers to re-fetch the input data from Mappers.

Algorithm 1 Word Count

```
1: Map(RowKey, RowResult)
2: for each user within a specific Internet supplier do
3:     emit(Internet − supplier, 1);
4: end for
5: Combine(Internet-supplier, list of values)
6: int_sum = 0;
7: for each value of list do
8:     int_sum = int_sum + value;
9: end for
10: emit(Internet − supplier, int_sum);
11: Reduce(Internet-supplier, list of int_sums)
12: fin_sum = 0;
13: for each value of list do
14:     fin_sum = fin_sum + value;
15: end for
16: emit(Internet − supplier, fin_sum);
```

Although MapReduce is widely used for large volume data processing, it has been sharply criticized by David DeWitt and Michael Stonebraker [DS08a, DS08b]. They considered MapReduce as "a major step backwards" and lack of novelty. The key comment for this argument is that MapReduce misses a lot of technologies which are *common* for today's database systems, e.g. high-level query language, indexing, transaction supporting,

integrity constraints, views, query optimization and so on. They wrote a paper [PPRea09] to compare the performance of MapReduce and a distributed database system. They concluded that distributed database systems beat MapReduce in many areas and show tremendous benefits.

However, the MapReduce advocates rejected such statements and point out that it is inappropriate to compare MapReduce with modern distributed database systems as they serve different purposes. They state that MapReduce is mainly designed for batch mode data processing and transformation based on a massive data volume. In consequence, technologies such as indexing, random data access and so on are unnecessary for MapReduce. Moreover, they claim that MapReduce is more scalable and can handle a higher data volume than any existing distributed databases.

2.3.2 Hive and Pig Latin

As we have already seen in the previous sections, users have to manually implement query processing when utilizing the MapReduce framework. This hand-coded data analysis has several limitations and constraints [TSea09, TSea10, ORea]:

1. *Requirement of programming skill.* MapReduce requires users to have the knowledge of programming languages, e.g. Java, C++ or Ruby. This limitation sets a huge obstacle to people who have no experience with programming and increases the learning curve when utilizing the MapReduce framework.

2. *Limited reuse and increased likelihood of errors.* As data processing is implemented manually, it is possible that two programmers write different MapReduce procedures to accomplish the same task. Moreover, there is increased likelihood that programmers write wrong or invalid statements during coding. In consequence, code-debugging is essential and cumbersome.

3. *Unreliable query optimization.* In general, query optimization is a key component for query processing, e.g. in SQL engines. However, for the MapReduce framework, this task is accomplished by applications. Or to be more specific, the programmers should decide which optimization strategy is more suitable in a certain situation. Generally, not all programmers are knowledgeable about database systems. Hence, it is

difficult for them to choose the right query optimization for a specific query. For example, to implement a join operation in MapReduce, we can implement it either as a single Map function or a combination of Map and Reduce. It depends on the size of input tables to decide which strategy to use. However, it is difficult for programmers to figure out what is "large" and what is "small". It is possible that one table is considered "large" when the capacity of work node is small. However, it can become "small" when more memory and larger disks are later added in the cluster. In consequence, the optimization of query processing is unreliable and depends on the experience and background of programmers.

4. *Hard documentation.* As data processing is implemented in Java or C++, it is mandatory for programmers to write comments or descriptions for their code. If someone writes "misleading" descriptions, this enforces people to dig into the code.

To avoid the aforementioned pitfalls, a lot of companies built their own high-level languages on top of MapReduce. The two most famous are *Hive* [TSea09, TSea10, LLea, Hiv] and *Pig Latin* [ORea, Pig]. Hive is an SQL-like declarative query language proposed by Facebook in 2009. Pig-Latin is a data-flow (procedural) query language which was introduced by Yahoo! in 2006. Both high-level languages support a large range of primitive types, e.g. integer, float or string and a set of complex types such as array, map and bag. Moreover, they contain a set of high level operators which can be used for query specifications.

Table 2.3 lists the high-level operators supported by Pig Latin and Hive. The first six operators have the same semantics as the typical relational operators. *Cogroup by* in Pig Latin is a new operator which takes more than one operands and performs join and group by at the same time. Different from Hive, Pig Latin supports *bag* ($multi - set$) type. When a bag type is used, a column can store multiple data values. However, as the data processing operators, e.g. filter, join and group by, can only be applied to the atomic attribute (flat relation), we need *Flatten* operator to unnest the nested attributes. $Multi - insert$ and *Split into* are two operations which can be used to insert data values into different outputs.

To operate with wide-column stores (WCSs), Hive and Pig Latin follow different strategies.

Hive	Pig Latin
Select (project)	Foreach generate
Filter	Filter by
Group	Group
Union	Union
Sort	Order by/Limit
Join	Join by
	Cogroup by
	Flatten
Multi-insert	Split into

Table 2.3: High-level operators of Hive and Pig-Latin

•Integrating WCS with Hive

Algorithm 2 Hive DDL

```
1: CREATE TABLE Web-surfing(key string, page string,
2: content string, supplier string, speed int)
3: STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
4: WITH SERDEPROPERTIES (
5: "hbase.columns.mapping" = ":key, Web:Page, Web:Content, Network:Supplier, Network:Speed");
```

Before processing data from WCSs, Hive first requires users to utilize a data definition languages (DDL) to specify the schemata of tables. Suppose we want to process the "Web-surfing" table (in Figure 2.1) by Hive. Algorithm 2 shows the appropriate DDL. Hive supports a default built-in function, namely, "HBaseStorageHandler" to map the table information between Hive and WCSs. When the following DDL expression is executed, it will only create a directory in the distributed file system and no data is yet loaded.

After schema information is specified, we can use the *INSERT OVERWRITE* command to load the data into Hive. Algorithm 3 shows an example of how we can use Hive to calculate the average Internet speed for a specific Internet supplier.

Algorithm 3 Calculating AVG of Internet speed by Hive

```
1: SELECT supplier,AVG(Speed) as Speed
2: FROM Web-surfing
3: GROUP BY supplier;
```

Note that, the execution of "HBaseStorageHandler" and "INSERT OVERWRITE" command will only load the current data versions without including timestamps. When users wish to load multiple data versions, a customized load function has to be manually coded and an array type should be used for each column.

•Integrating WCS with Pig Latin

Algorithm 4 Calculating AVG of Internet speed by Pig Latin

```
1: Web = LOAD 'hbase://Web-surfing'
2: USING org.apache.pig.backend.hadoop.hbase.HBaseStorage(
3: 'Web:Page, Web:Content, Network:Supplier, Network:Speed', '-loadKey
   true -limit 5')
4: AS (id:string, page:string, content:string, supplier:string, speed:int);
5: Groups = GROUP Web BY supplier;
6: Average = FOREACH Groups GENERATE group, AVG(speed);
```

Like Hive, Pig Latin supports default built-in functions for WCS. However, different from Hive, schema information is indicated when data is processed, namely, data definition language and data manipulation language exist in a single script. Algorithm 4 shows a Pig Latin example which calculates the average speed for a specific Internet supplier. The data processing is represented as a data flow in which users have to understand both input and output schemata for each statement. For example, *GROUP BY* will generate a default *group* attribute for the output data.

Again, the default built-in function will only load the latest data value for each column. When multiple data versions are required, a customized load function has to be provided.

2.3.3 Pitfalls and limitations

As described previously, *Hive* requires users to first define the table schema before any data can be loaded into the table. This strategy is sometimes called "schema on write". The main benefit of this approach is that Hive can check the type of each column when loading data and make certain optimizations at query compile time. However, the requirement for a fixed schema contradicts one core property of NoSQL stores, namely, schema flexibility. Moreover, Hive gives users an illusion that the query is executed where the data is located and no data transfer occurs between Hive and WCSs. Nevertheless, the truth is that Hive's table-creation command will generate a file directory in the distributed file system and load the data from WCS to Hive. In consequence, Hive's query will only be performed on the copy of WCSs instead of real data. This strategy may be a problem when the WCS is frequently modified. The only solution to this problem is to refresh the copies in Hive. Obviously, when the size of extracted tables is very large, this loading process will be very long. Even worse, Hive utilizes *overwrite-loading* instead of *incremental loading*. The overwrite-loading will overwrite everything even when only a small part has been changed.

Pig Latin requires users to define the structure of data before any data can be processed. However, recently, the Pig Latin community introduced some new advanced features and claims that it is unnecessary to explicitly define meta-data information for query processing. Users can utilize positional notation, e.g. 0, to reference attributes in query processing. Even more, Pig Latin has the ability to guess the type of an attribute when it is not explicitly denoted. Although this strategy increases programming flexibility, Pig Latin cannot guarantee the "correctness" of query results. The term "correctness" means the desired or expected query results. Suppose we store the date attribute as a string but utilize it to compare with an integer, e.g. $date \geq 4$. When the date type is not explicitly specified, Pig Latin will try to interpret date as an integer instead of string. Obviously, it has a high possibility that unexpected query results will be generated.

Different from Hive, Pig Latin combines data definition statement and data manipulation statement in the same script. This strategy is also called "schema on read". The main benefit of this approach is that the schema only needs to be defined through the data processing. Nevertheless, one big problem of this approach is that data definition language must appear in every Pig Latin script and different programmers may point to the same data sets

with different notations. Hence, it is hard to detect the similarities between different Pig Latin scripts and program re-usability is therefor decreased.

2.4 Summary

In this section, we first reviewed the NoSQL history and explained the meaning of "NoSQL". NoSQL stores can be seen as complementary to relational database systems (RDBMSs). Different from RDBMSs, NoSQL stores have limited SQL support. Data is partitioned and distributed in the cluster. In general, they support high scalability and robust fault tolerance.

Wide-column stores (WCSs) are one kind of NoSQL stores. They introduce new data models and properties, such as column-family, data versioning and time-to-live constraints. Column-family can be viewed as column prefix and the columns which belong to the same column-family are stored contiguously on disk. Data versioning means that one column can hold multiple data versions which are sorted by timestamps. Time-to-live is a property which indicates how long a data version is kept in the system. Different from traditional DBMS, WCS only support simple client APIs and single-row transactions (or tunable data consistency).

As WCS lack SQL engines, users can write either low-level programs for query processing, such as MapReduce procedures, or exploit high-level languages, such as Pig Latin or Hive. Pig Latin and Hive are built on top of MapReduce and try to facilitate query specification and avoid certain pitfalls and restrictions existing in the MapReduce framework.

Chapter 3

Temporal data management and processing – state of the art

Temporal data modeling and temporal data processing has been extensively researched for several decades and a lot of approaches have been proposed.

In this chapter, we first describe how temporal data can be modeled in the *point-based view of temporal data model* [BOea98, BBJ] and the *interval-based view of temporal data model* [DBS96, SAea94, JSS94, JSS, TG89, Tan97]. Although the point-based temporal data model is closer to how we model time, it is not as widely used as the interval-based temporal data model. When utilizing the *interval-based temporal data model*, two approaches can be adopted. The first one is to append each table with two additional columns to denote a validity interval (*tuple time-stamping temporal model* [DBS96, SAea94, JSS94, JSS, KeM11]). The second one is to associate each attribute with a temporal interval (*attribute time-stamping temporal model* [TG89, Tan97]). The tuple time-stamping model is currently supported by the SQL standard and most commercial database vendors, such as IBM, Oracle or SAP.

To process temporal data, two main approaches can be followed:

- The first one is to transform temporal data and temporal queries into their non-temporal counterparts [DBG12, SBea97]. This approach treats the temporal attributes (in which time information is stored) as non-temporal (or normal) attributes and temporal queries which are applied

on temporal data will be translated into non-temporal queries based on a set of transformation rules.

- The second one is to directly manage and process temporal data without auxiliary temporal query and temporal data transformations [DBS96, SAea94, JSS94, JSS, TG89, Tan97, KeM11].

In the following sections, we will introduce the approaches mentioned above and discuss their advantages and pitfalls.

3.1 Various temporal data models

In this section, we will give an overview of how temporal information can be modeled. We first introduce the *point-based model* in which each tuple or each individual attribute value is associated with a timestamp (TS). Then, we describe the *interval-based model* in which each tuple or each individual attribute value is associated with a temporal interval.

3.1.1 Point-based model

In the real world, *time* is generally considered as an infinite *continuous* stream. However, in the scientific world, for simplicity, time is usually modeled as an infinite *set* which consists of *discrete* time instants with a total order. Each element in such a set is assigned with a non-negative integer.

In the non-temporal database context, an entity can be modeled by either *first-normal-form* ($1NF$) [Cod70][1] or *non-first-normal-form* (NF^2) [Mak77]:

- A relation (table) is in $1NF$ if the domain of each column contains only atomic values, and the value of each attribute contains only a single value from that atomic value domain.

- A relation (table) is in NF^2 if the domain of each column can be non-atomic, e.g. set, bag or array, and each column can therefor store multiple complex values.

[1] $2NF$, $3NF$ and $BCNF$ (Boyce-Codd Normal Form) are considered as refinements of $1NF$

Salaries

Name	Salary	Month
Hans	1000	1
Hans	1000	2
Hans	1000	3

1NF

Salaries

Name	Salary
Hans	(1000,1) (1000,2) (1000,3)

NF^2

Figure 3.1: Point-based model with $1NF$ and NF^2

In consequence, to model temporal data, we can either utilize $1NF$ in which the whole tuple is appended with a timestamp (TS) or use NF^2 in which TS is embodied in each attribute value. Figure 3.1 shows examples when utilizing $1NF$ and NF^2, respectively. Suppose the "Salaries" table stores the salaries for employee "Hans" from first month to third month. When $1NF$ is used, three tuples should be stored. Nevertheless, when NF^2 is used, only one tuple will be stored in which the "Salary" column contains three data values. Obviously, the NF^2 model maintains less data than $1NF$, e.g. string "Hans" exists three times when $1NF$ is used.

Although the point-based model is simple and straightforward, it has several pitfalls:

- *Data redundancy.* Back to the previous example, if we want to store the annual salary for "Hans", 12 tuples have to be inserted into the "Salaries" table. Otherwise, we have to change the granularity of time, from month to year.

- *Limited expressiveness.* Suppose we wish to denote a data item is "infinitely" valid. As the point-based model stores valid data for each time instant, it is impossible to represent a time-point for infinity. One can argue to use ∞ as a symbol to denote infinity. However, it is still unclear how to calculate the number of time points between a specific starting point and infinity.

3.1.2 Interval-based model

To overcome the limitations and constraints caused by *point-based model*, a new temporal model called *interval-based model* is proposed [DBS96, SAea94, JSS94, JSS, TG89, Tan97, KeM11]. In contrast to the point-based model, the interval-based model appends two attributes for each data item to form the start- and end-time point to indicate how long the data item is valid.

Salaries

Name	Salary	Start	End
Hans	1000	1	3
Hans	1500	4	7
Joe	2000	5	12

1NF

Salaries

Name	Salary
Hans	(1000, [1,3]), (1500, [4, 7])
Joe	(2000, [5,12])

NF^2

Figure 3.2: Interval based model with $1NF$ and NF^2

In analogy to the point-based model, a temporal interval (TI) can also be introduced in $1NF$ or NF^2. In most research papers, the first option ($1NF$ + TI) is usually called *tuple-timestamping* model [DBS96, SAea94, JSS94, JSS, KeM11] and the second one (NF^2 + TI) is named *attribute-timestamping* model [TG89, Tan97].

Figure 3.2 shows the examples for the interval-based model with $1NF$ and NF^2, respectively. The "Salaries" table in $1NF$ now contains two columns, namely, "Start" and "End" to denote how long a specific salary for an employee is valid. For the NF^2 relation, the "Salary" column consists of a pair (*value*, *temporal interval*). Such a pair is also called "*temporal atom*" [TG89, Tan97] which is treated as an atomic data unit (analogy to string, float, integer and etc. in relational database systems).

In recent years, most database vendors, such as IBM, Oracle and Microsoft as well as the SQL standard [KeM11] have picked up the capability to support temporal data management and processing. They utilize the tuple-timestamping model and refine the semantics of time into two different

categories:

- *Valid time* (also known as *application time*) is a time period during which a tuple (fact) is (or was) true in the real world. The value of valid time can be assigned or changed by users and tuples can be updated and overwritten.
- *Transaction time* (also known as *system time*) is a time period during which a tuple (fact) stored in database systems is considered to be true or current. In contrast to valid time, users are not allowed to modify the time information. Instead, values of transaction time are automatically generated by the database system itself. Moreover, a delete operation will not discard a tuple but update the interval end of that tuple. An update operation will cause a temporal interval modification of the current row and the generation of a new row. Figure 3.3 shows this example. Suppose we update the salary of tuple "Joe" (in Figure 3.2) from 2000 to 2500 at time instant 8. The update operation will cause two data modifications, namely modifying the TI-end of updated tuple (from 12 to 8) and inserting a new tuple "(Joe, 2500, 8, 12)".

Salaries

Name	Salary	Start	End
Hans	1000	1	3
Hans	1500	4	7
Joe	2000	5	8
Joe	2500	8	12

1NF

Figure 3.3: Update Salaries table with transaction time

When a table maintains both *Valid time* and *Transaction time*, it is called "Bi-temporal relation".

Although interval-based model is widely used, it has several limitations and constraints:

- It is expensive to detect data redundancy, e.g., two tuples may share the same non-temporal attributes but have temporal period overlapping.
- Detecting conflicting data is complicated and cumbersome, e.g., two "versions" (tuples) which represent the same "object" may have the time period overlapping but with different non-temporal attributes.

3.2 Temporal data processing

In this section, we mainly focus on how to query and process temporal data. As the technologies of non-temporal database systems e.g. relational databases are mature, the first option is to transform the temporal data and temporal query into their non-temporal counterparts. Although this approach can be seamlessly integrated with the existing database systems, the query performance may not be desirable. In consequence, attempts are made to build temporal operators and corresponding temporal query engines to directly process temporal data.

3.2.1 Processing temporal data with non-temporal operators

As the theories and technologies of non-temporal databases, e.g. relational databases, are extensively studied and increasingly mature, some approaches extend the relational databases with additional functionality to support temporal data processing [DBG12, SBea97]. This approach generally contains three steps:

1. Transforming temporal query and temporal data into their non-temporal counterparts;
2. Applying non-temporal operators into transformed non-temporal data;
3. Rebuilding temporal query results based on non-temporal query results.

To process temporal data with non-temporal operators e.g. relational algebra, the first task is to transform the temporal data into non-temporal data. To guarantee the correctness of temporal query processing, people proposed a notion called *Snapshot Reducibility* [SBea97, DBG12]. When Snapshot Reducibility is used, temporal data will be translated into non-temporal data

at each individual time point by discarding the temporal information. This approach works fine when predicates and functions do not include temporal comparisons. However, as temporal information is discarded, it is impossible to rebuild the temporal relation after query processing.

Extended Snapshot Reducibility [LO09] is an extension of Snapshot Reducibility in which temporal information is kept in each snapshot but will be treated as non-temporal attributes. To achieve that, authors in [DL02, LO09] proposed two operators *UNFOLD* and *FOLD*. The UNFOLD operator will transform each interval-based tuple into point-based tuple where FOLD collapses value-equivalent tuples over consecutive time points into interval-based tuples over maximal interval length. Two tuples t_1 and t_2 are *value-equivalent* if the non-temporal attributes of t_1 and t_2 share the same values. Authors in [DBG12] proposed two temporal primitives, i.e. *Temporal Splitter* and *Temporal Aligner* to accomplish the same task. Different from the approaches described in [DL02], [DBG12] exploits temporal intervals as the intermediate temporal information representation instead of time points.

To transform temporal queries into their non-temporal counterparts, a set of transformation rules are proposed by [DBG12] based on the relational algebra. However, such rules are not straightforward and can generate very complex non-temporal queries after query transformation.

Although processing temporal data based on non-temporal operators can be seamlessly integrated with existing database systems, the query performance can be heavily decreased when the volume of temporal data is large (as it increases the time for temporal data and query transformation).

Note that, as relational databases are usually utilized as query processing engines, the *tuple-timestamping* model is mainly adopted for this approach.

3.2.2 Processing temporal data with temporal operators

To avoid temporal query and temporal data transformation, researchers proposed a set of temporal operators which can be directly applied to temporal data [DBS96, SAea94, JSS94, JSS, TG89, Tan97, KeM11]. In this section, we first give a short overview of temporal relational algebra which is an extension of relational algebra. Then, we introduce how to handle the *attribute-timestamping* model.

Processing tuple-timestamped relations

To process relations in the *tuple-timestamping model*, the relational algebra is extended to temporal relational algebra [DBS96, SAea94, JSS94, JSS, KeM11]. In contrast to the non-temporal algebra, temporal relational algebra introduces several new notions, e.g. "*value-equivalent tuples*" and "*temporal duplicates*", to guarantee the correctness of temporal query processing and a set of syntactic sugar, e.g. *meet* and *overlap*, to facilitate temporal query specification.

We give a short description of each temporal relational operator as follows. Please note that the semantics of each temporal operator is based on the temporal language *TSQL2* [SAea94, JSS94] and its extensions [JSS, DBS96].

- *Project* $\pi^{tt}_{(A,TI)}(R)$ outputs the desired attribute values from temporal table R which are specified in A and are valid during TI.
- *Filter* $\sigma^{tt}_{p}(R)$ selects tuples in R which satisfy the filter predicate p, where p can include temporal and non-temporal comparisons. To simplify the specifications of temporal comparisons, syntactic sugar in term of temporal predictions is defined, e.g. "*after*", "*precede*".
- *Natural Join* $R_1 \bowtie^{tt}_{p} R_2$ joins tuples from two temporal tables R_1 and R_2 in which they satisfy the join predicate p and are valid during the same period of time.
- *Union* $R_1 \cup^{tt} R_2$ combines two temporal tables and coalesces tuples which are value-equivalent and their TIs are overlapping or adjacent.
- *Difference* $R_1 -^{tt} R_2$ outputs tuples in R_1 which do not "exist" in R_2. When value-equivalent tuples are found in both R_1 and R_2, the TI of result tuples is the difference of TIs of the value-equivalent tuples.
- *Intersect* $R_1 \cap^{tt} R_2$ generates tuples from R_1 and R_2 in which they are value-equivalent and are valid during the same period of time.
- *Cartesian-product* $R_1 \times^{tt} R_2$ concatenates each tuple in the first relation with every tuple in the second relation which are valid during the same time period.

Notice that, the $\pi^{tt}_{(A,TI)}(R)$ operator does not eliminate duplicates or combine tuples which are value-equivalent and have TIs which are overlapping or

adjacent. Instead, users need to utilize a "*Coalesce*" [BSS96] operator to do this task. Temporal operators such as "*Splice*", "*Snapshot*" and "*Drop-time*" [SAea94] are not described, as they can be considered as syntactic sugar and can be represented by the temporal operators described above. For example, operator $Snapshot(R, t)$ can be represented as $\pi^{tt}_{(A,TI)}(R)$ in which A contains all non-temporal attributes of table R and TI is formed as $[t, t+1)$. $[t, t+1)$ is a interval representation of time instant t.

Different from the non-temporal aggregation, various forms of temporal aggregations can be defined:

1. *Instantaneous temporal aggregation* [KS95] in which the time-line is partitioned based on the finest time granularity supported by the temporal database and the aggregation value at each time t is calculated by all the tuples which are valid at time point t.

2. *Moving-window or cumulative temporal aggregation* [SGM93] in which the time-line is partitioned based on the finest time granularity supported by the temporal database and the aggregation value at each time t is calculated by all the tuples which are valid during $[t - m, t]$ where m denotes the length of the moving window.

After partitioning the tuples, a set of aggregation functions, such as MIN, MAX, AVG can be applied to each individual group.

Processing attribute-timestamping relations

To process the attribute-timestamping model, two approaches can be adopted.

- The first one is to translate the *attribute-timestamping model* into the *tuple-timestamping model* and reuse the temporal relational algebra.

- The second one is to directly define nested temporal operators for the *attribute-timestamping* model.

For the first approach, [TG89, Tan97] define a series of temporal table and temporal atom transformation functions, namely, *Unnest* ν and *Temporal Atom Decomposition* ϱ to translate the attribute-timestamping model into the tuple-timestamping model. Then, temporal relational algebra can be applied to the tuple-timestamping tables. Finally, functions such as *Nest* μ

and *Temporal Atom Formation* τ can be exploited to rebuild the attribute-timestamping tables.

For the second approach, we describe a set of nested temporal operators based on [Tan97]:

- Project $\pi^{at}_{(A,TI)}(R)$ outputs attribute values specified in A that are valid during TI.
- Filter $\sigma^{at}_{p}(R)$ outputs the tuples which satisfy p. Note that, a complete tuple will be generated even if only a subset of that tuple satisfies p.
- Union $R_1 \cup^{at} R_2$, Difference $R_1 -^{at} R_2$ and Intersect $R_1 \cap^{at} R_2$. These three operators are defined as same as their relational algebra counterparts, i.e. no temporal semantics is supported.
- Cartesian-product $R_1 \times^{at} R_2$ concatenates each tuple in the first relation with every tuple in the second relation. Different from $\times^{tt}$ described in the previous section, $\times^{at}$ does not require that all the data versions are valid during the same time period.
- Join $R_1 \bowtie^{at}_{p} R_2$ is represented as a combination of Cartesian-product $\times^{at}$ and Filter σ^{at}_{p}.

To the best of our knowledge, there exists no proposals which specify how to define temporal aggregation based on the attribute-timestmaping model until now.

3.3 Summary

In this chapter, different models for representing temporal data, namely, *point-based model* and *interval-based model* are introduced. For each individual temporal model, we can further refine it by distinguishing whether the temporal information is attached to the whole tuple (*tuple-timestamping model*) or each individual attribute (*attribute-timestamaping model*).

For processing temporal data, two ways can be followed. The first approach is to translate temporal queries and temporal data into their non-temporal counterparts and reuse technologies for non-temporal data processing, e.g. approaches and optimization strategies utilized for relational database systems. Although this approach can be seamlessly integrated with

existing database systems, transforming temporal queries is not a trivial task and temporal data transformation can heavily decrease query performance.

To avoid the transformation of temporal queries and temporal data, researchers proposed a set of temporal operators which can be directly applied to temporal data. Temporal relational algebra is an extension of relational algebra with support for *tuple-timestamping model*. When the *attribute-timestamping model* is used, a general approach is to first translate *attribute-timestamping model* into the *tuple-timestamping model* and then utilize the temporal relational algebra. After temporal processing is finished, original table structure (*attribute-timestamping model*) will be rebuilt. Although there exist several proposals for nested temporal operators, they reuse the set operators such as *Union*, *Difference*, *Intersect* and *Cartesian-product* which ignore the temporal semantics kept in each tuple.

Chapter 4

Managing temporal data in wide-column stores

Recently, it has become increasingly popular to build data warehouses based on wide-column stores (WCSs). Besides storing the current data, a data warehouse also maintains historical data for the purpose of decision support, controlling and revision capabilities.

In this chapter, we introduce how temporal data can be maintained in WCSs. Although each data version in a WCS contains a corresponding timestamp (TS), its semantics is ambiguous. In consequence, we first illustrate the meaning of TS. As we have already described in Chapter 2, each column in a WCS can maintain multiple data versions. However, its default temporal interval (TI) representation can lead to wrong or misleading temporal query results. To overcome such problems, we propose two alternative temporal interval representations, in which each tuple (or data version) contains an explicit TI. Finally, we describe how these different representations can be transformed to each other.

4.1 Semantics of timestamp

Before discussing the semantics of timetamp (TS), we first present a formalization of WCSs.

A schema R for a WCS table is a collection of columns of the form $R = (rk, CF_1{:}Col_{11}, \ldots, CF_n{:}Col_{nn})$, where rk is shorthand for row key and the subscript n denotes the number of column families (CFs). Each CF_i is composed of a set of columns $(Col_{i1}, \ldots, Col_{ij})$. The value of a column is a set

of data versions in which each data version D_m can be further decomposed as a pair ($Value$, TS). $Value$ denotes the content of D_m and TS denotes when that data version is generated. For each column, TS functionally determines the Value, i.e. $TS \longrightarrow Value$.

A WCS table r is an instance of a WCS schema R. $Dom()$ is a function which maps an attribute name into its value domain. In the WCS context, $Dom(rk)$ has usually a string type. $Dom(Value)$ can be any set of atomic values, such as integer, float, string and etc. $Dom(TS)$ is assigned as a discrete time domain which consists of a set of long nonnegative integers with an ascending order. $Dom(CF : Col) = Dom(Value) \times Dom(TS)$ and $Dom(R) = Dom(rk) \times Dom(CF1 : Col_{11}) \times \ldots \times Dom(CFn : Col_{nn})$, where $\times$ is Cartesian product. Moreover, a so-called time-to-live (ttl) property can be specified based on each CF to denote how long a data version can exist in that CF. When ttl is not specified, its default value is infinity ∞.

∞ is a symbol to denote a data version (or a tuple) always "alive". Its value can be set to either NOW [SAea94, YW98] or the maximal time instant based on the time granularity supported by the temporal table [KeM11]. NOW is a symbol which represents the current time and its value automatically increases when time elapses. SQL:2011 standard chooses the second option, e.g. the value of "Sys_End" column in a *System-versioned table* is set to "9999-12-31 23:59:59". For representing ∞ in WCSs, we also choose the second option. Although we have chosen the same strategy as SQL standard, the value of ∞ varies based on the finest time granularity supported by the WCS table. For example, when the finest time granularity of a table is $YEAR$, ∞ has value "9999".

Network-speed

RK	Network:Supplier	Network:Speed
Tom	1&1: 5 Telecom : 1	1270K : 5, 920K : 4, 1115K : 1

Figure 4.1: WCS example

Clearly, a WCS table does not satisfy $1NF$, as the attribute type of each tuple is not atomic. However, different from the general NF^2 relations, the nesting level of a WCS table is fixed, i.e. 1 (we view the nesting level of $1NF$ as 0). In the following, we use $t[S]$ to denote the value of navigation path S of a tuple t and $Attr(A)$ to indicate a set of attributes that belong to

A, where A can be a table name, a column-family name or a column name. Corresponding example is shown in Figure 4.1.

Due to the previous description, WCSs follow an *attribute-timestamping* approach by attaching a TS to each data version. However, in contrast to temporal databases, the explicit TS just represents the start of a time interval. The TI is only implicitly represented when we assume the end of the interval to be determined by the start-TS of the subsequent version (or ∞ for the most recent version), which is consistent with the semantics of version timestamps. This interpretation constrains the derived TIs belonging to the same column to form a contiguous time interval, e.g. the TIs of data versions "Telecom:1" and "1&1:5" in Figure 4.1 are [1, 5) and [5, ∞), respectively. In addition, the time interval for the most recent version is also limited by the ttl property. For example, suppose we set the ttl property for column family "Network" to 10. Although "1&1:5" is the latest data version, its TI is [5, 15). As the TI for each data version is implicitly represented inside the column, we call the original table representation *implicit history representation* (IHR).

Furthermore, another question in terms of the semantics of TS in WCSs arises. In the temporal database literature, there are two orthogonal time dimensions: 1) *Valid time*, which indicates the time interval during which a data value reflects the state of the real world; 2) *Transaction time*, which denotes when a data item is recorded in the database. Valid time and transaction time are usually depicted as a time period [t_1, t_2) which denotes that a data value holds at time t where $t_1 \leq t < t2$. The valid time can be assigned and modified by users, whereas the transaction time is generated and maintained automatically by the database system.

In WCSs, due to the different usages of the *Put* and *Delete* commands, the TS can be either arbitrarily specified by users or automatically generated by the system. If the TS is denoted by users, this implies that any data versions can be inserted or discarded at any point of time in the version history of each column. Consequently, the *Put* and *Delete* commands with the explicit TS assignments may cause TI modifications of existing data versions (not only the latest data version). For example, in Figure 4.1 if a data version "800K:3" is inserted between versions "1115K:1" and "920K:4", the TI for "1115k:1" is implicitly changed to [1, 3). In this situation, the TS in WCS has *valid time* semantics.

However, when TS is automatically generated by WCS, 1) for the *Put* command, the TS of new generated data version will be greater than all the existing data versions; 2) for the *Delete* command, either the current

(latest) data version will be deleted (data version deletion) or the whole column/column-family will be discarded (column and column-family deletions). Hence, either only the current data version will be changed (*Put* command and data version deletion) or all the data versions will be eliminated (column and column-family deletions). For example, if a data version "800K:8" (where TS 8 is generated by the WCS) is inserted into "Network:Speed" column in Figure 4.1, the TI of "1270K:5" is changed from [5, ∞) to [5, 8) where the TI for "920K:4" and "1115K:1" are still [4, 5) and [1, 4), respectively. In this situation, TS in WCS is close to *transaction time*.

Hence, the temporal semantics of TS in WCS is ambiguous, namely, it can be understood as either *valid time* or *transaction time* based on the usages of *Put* and *Delete* commands. Consequently, the user or application has to make consistent use of temporal concepts supported by WCSs. The TS for a single column should have the semantics of either valid time or transaction time but not both. Moreover, if *Bi-temporal* data needs to be maintained (i.e., both transaction and valid time data are needed), additional columns need to be added by the application to keep the time information. For the usage of WCS, we assume that the application is aware of this.

4.2 Drawbacks of default temporal data representation

Although the implicit history representation (IHR) supported by WCS is suitable for data storage, it can cause wrong or misleading results during query processing.

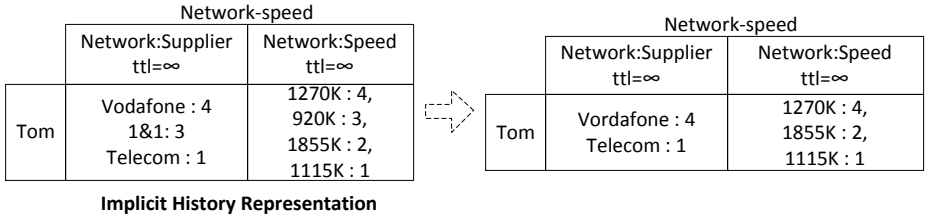


Figure 4.2: Select network supplier whose speed is faster than 1000K by using IHR

Suppose we use the "Network-speed" table in Figure 4.2 as an input and wish to select Tom's Internet suppliers whose speed has at some points been faster than 1000K. The filter operation will discard "1&1:3" in the "Supplier"

column and "920K:3" in the "Speed" column. The right-hand side shows the filter results. As the TI for each data version is implicit, directly discarding data versions will cause TI changes of the remaining data versions, e.g. the valid TI of "Telecom:1" is changed from [1, 3) to [1,4). Obviously, this produces incorrect results.

Besides causing the misleading temporal query results, IHR also has limited ability for modeling temporal data. For example, it is impossible for IHR to store the real world events which occur in the future.

4.3 Two explicit temporal interval representations

To solve the issues described in the previous section, we can directly associate each data version with an explicit TI. The TI of each data version is deduced from the column as $[TS_c, TS_s)$, where TS_c denotes the timestamp of that data version and TS_s indicates the TS of its successor. Figure 4.3 shows the equivalent representation of the "Network-speed" table on the left and the correct results of filter processing on the right. We call this new table representation *explicit history representation* (EHR).

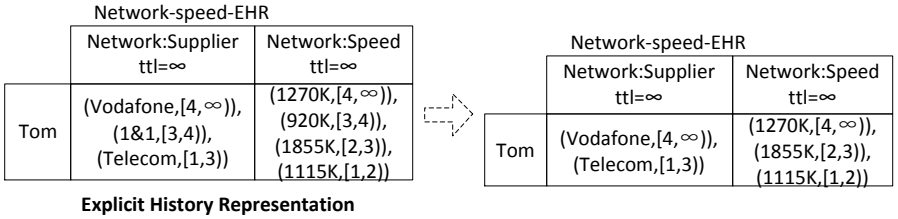


Figure 4.3: Select network supplier whose speed is faster than 1000K by using EHR

In contrast to the original data version definition, we model a data version D in the WCS as a pair ($Value$, TI) where:

- $Value$ indicates the value of D;
- TI denotes how long D is temporally valid and has the form [$Start$, End).

As an alternative to grouping multiple data versions with explicit TIs in a single column, we can also adopt the *tuple-timestamping* approach by splitting each IHR tuple into several tuples in which each column contains only

single data version and the row key includes the valid TI to guarantee its uniqueness. We call this table representation *tuple-timestamping representation* (TTR). Figure 4.4 shows the TTR example derived from "Network-speed" on the left and the correct results for filter processing on the right. For better readability, we specify the row key in TTR as a pair (srk, TI) where srk denotes the original row key value extracted from the corresponding IHR table and TI indicates the valid time interval which has the form $[Start, End)$.

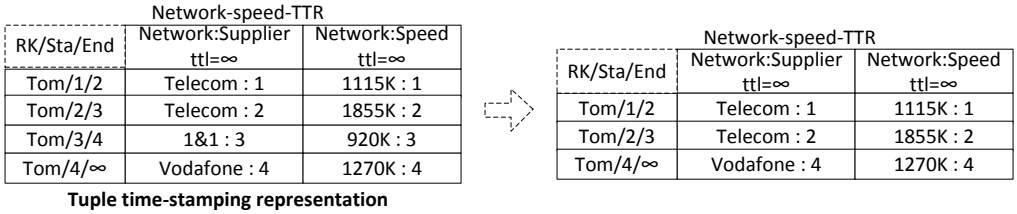


Figure 4.4: Select network supplier whose speed is faster than 1000K by using IHR

Although both TTR and EHR can guarantee the soundness of query processing, each of them has drawbacks. For storing TTR tables in WCSs, the TI has to be encoded in the row key to guarantee its uniqueness, as composite keys are not supported by WCSs. Moreover, this strategy can also cause significant data redundancy. For example, in Figure 4.4, the strings "Tom" and "Telecom" appear three times and twice, respectively. As WCSs usually manage a tremendous volume of data, the TTR tables will need large disk capacity.

EHR has an optimal structure for data storage, but its physical representation in WCSs is very complicated. For example, in HBase, we have to "encode" the pair ($Value$, TI) as the data value (e.g. JSON string) and $TI.Start$ as the TS for each data version. Moreover, when a data version satisfies a filter predicate, to generate a complete row, the TI of that data version will be utilized as a selection criterion to fetch the corresponding data versions from the other columns in which their TIs have overlapping. Hence, the data processing tasks for EHR will be more complicated and need more time to "extract" the actual data values compared to TTR.

In consequence, choosing the table representation for temporal query processing is a trade-off between data capacity and data processing complexity. We study the influence of temporal query performance based on various WCS table representations with their corresponding temporal operators in Section 5.4.

4.4 Transformations between three temporal representations

Based on the previous descriptions, an IHR table has to be translated into an EHR or a TTR table when the correctness of temporal query processing needs to be guaranteed. To model the same type of temporal entities, an EHR table maintains less data redundancy than a TTR table. However, as the data model supported by TTR is simpler than EHR, complicated temporal queries can be easily defined and implemented by using TTR representation. In this section, we study how these three different table representations can be transformed into each other.

We define 4 table transformation operations which transform IHR to EHR (T_{IE}), IHR to TTR (T_{IT}), EHR to TTR (T_{ET}) and TTR to EHR (T_{TE}), respectively. In the following algorithms, we use function *emit* to insert data values in WCS. In real applications, "emit" can be replaced by "Put" command (for HBase) or "Set" (for Cassandra).

T_{IE}:

Algorithm 5 Transforming IHR into EHR

```
1: Input: IHR table
2: Output: EHR table
3: Map(RowKey, RowResult)
4: for each data version D_n in a column do
5:     if D_n is not current then
6:         D_n.TI = [D_i.TS, D_j.TS), D_j is the successor of D_i ;
7:         emit(RowKey, D_n)
8:     else
9:         if ttl ≠ ∞ then
10:            D_n.TI = [D_i.TS, D_i.TS + ttl);
11:        else
12:            D_n.TI = [D_i.TS, ∞);
13:            emit(RowKey, D_n)
14:        end if
15:    end if
16: end for
```

T_{IE} takes an IHR table as an input and outputs its corresponding EHR

table. The explicit TI of a data version D_n in an EHR column is derived from its corresponding data version D_i in the IHR column and formed as $[D_i.TS, D_j.TS)$, where D_j is the immediate successor of D_i. When D_i is the current data version, its end point of TI is either denoted by ∞ or calculated by using ttl. Algorithm 5 describes how T_{IE} can be implemented using MapReduce.

T_{ET}:

Algorithm 6 Transforming EHR into TTR

```
1: Input: EHR table
2: Output: TTR table
3: Map(RowKey, RowResult)
4: SortedList TSL;
5: for each data version D_n in a column do
6:     TSL.add(D_n.TI.Start);
7:     TSL.add(D_n.TI.End);
8: end for
9: for each element TS_i in TSL do
10:    if TS_i is not the last element then
11:        TI = [TS_i, TS_j), where TS_j is the successor of TS_i;
12:        for each data version D_n in a column do
13:            if D_n.TI ∩ TI ≠ ∅ then
14:                emit(RowKey+TI, D_i);
15:            end if
16:        end for
17:    end if
18: end for
```

T_{ET} takes an EHR table as an input and outputs its corresponding TTR table. For every tuple in EHR, T_{ET} will first collect the TIs (start time point and end time point) of all data versions, and then the TI which is formed by two adjacent time points will be exploited as a derived TI. Finally, the derived TI will be utilized as a selection criterion to select the data versions from EHR columns. The corresponding algorithm is shown in Algorithm 6.

We illustrate the T_{ET} operation in Figure 4.5. The TI set of row "Tom" in EHR is represented graphically at the top right corner in which the time point set consists of 1, 2, and 3. The corresponding derived TIs are indi-

cated at the bottom right which are $[1, 2)$ and $[2, 3)$, respectively. T_{ET} then exploits each derived TI ($[1, 2)$ or $[2, 3)$) as a selection criterion to scan both "Network:Supplier" and "Network:Speed" columns to find the matching data versions. The resulting TTR is shown at the bottom left.

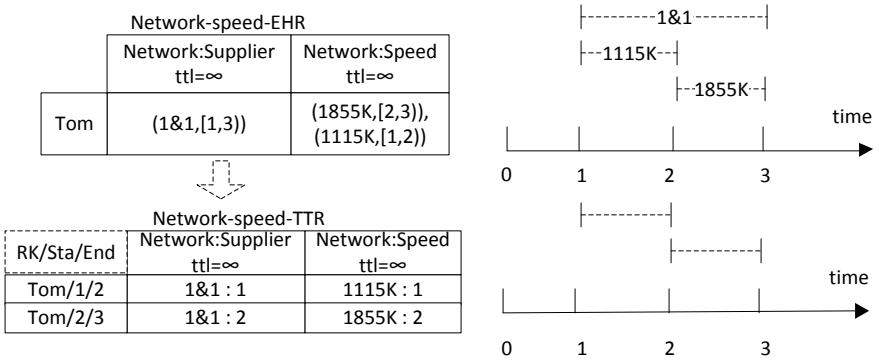


Figure 4.5: Transforming EHR to TTR

T_{TE}:

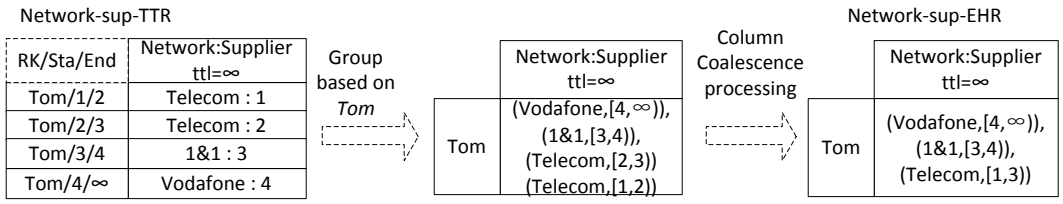


Figure 4.6: Transforming TTR to EHR

T_{TE} takes a TTR table as an input and outputs its corresponding EHR table. T_{TE} first groups the TTR tuples which share the same $rk.srk$ together. At the same time, the TI in the row key will be attached to each data version. At last, several data versions that have the same value will be coalesced into a single data version when their TIs are overlapping or adjacent. Figure 4.6 shows an example of T_{TE}. Two arrows indicate the data processing tasks described above.

Algorithm 7 describes how to transform a TTR table into an EHR table.

Algorithm 7 Transforming TTR into EHR

```
1: Input: TTR table
2: Output: EHR table
3: Map(RowKey, RowResult)
4: for each data version D_n in a column do
5:     Meta = RowKey+Column-family+Column;
6:     emit(Meta, D_n+TI);
7: end for
8: Reduce(Meta, List)
9: Map<Value, SortedList> maps;
10: for each element D_i in List do
11:     maps.keySet.add(D_i.Value);
12:     maps.SortedList.add(TI);
13: end for
14: for each SortedList in maps do
15:     TI_new = Compact TI from SortedList;
16:     emit(Meta, D_n + TI_new);
17: end for
```

T_{IT}:

T_{IT} takes an IHR table as an input and outputs its corresponding TTR table. T_{IT} can be represented as a T_{IE} operator followed by a T_{ET} operator. Note that, these two-phase transformations are only used to facilitate introducing the functionality of T_{IT}. In the real implementation, an IHR table will be directly transformed into a TTR table.

Due to the descriptions of table transformations, each IHR table can be mapped to one EHR table and one TTR table. Moreover, one EHR table can be mapped to one TTR table and vice versa. We omit to define the EHR to IHR and TTR to IHR transformations, as not every EHR table or TTR table can be transformed back to IHR. We utilize the EHR table at the right-hand side in Figure 4.3 as a counter-example, as the TIs of column "Network:Speed" do not form a contiguous time interval. It is hence impossible to rebuild the corresponding IHR. The same counterexample for TTR can be found at the right-hand side in Figure 4.4.

The reasons for inapplicable transformations (EHR to IHR or TTR to IHR) are due to the characteristics of IHR, namely,

1. The valid temporal interval for each column can only be $[OSta, \infty)$ or $[OSta, LSta + ttl)$, where $OSta$ and $LSta$ denote the start-time of the *oldest* data version and the start-time of the latest data version in that column, respectively;

2. The TI for each data version among the same column has to be contiguous, namely, for any two data versions D_1 and D_2 in an IHR column, if D_2 is the immediate successor of D_1, it denotes $D_2.TI.Start = D_1.TI.End$.

In consequence, any EHR or TTR table can be transformed to its corresponding IHR table if and only if the aforementioned two conditions are satisfied.

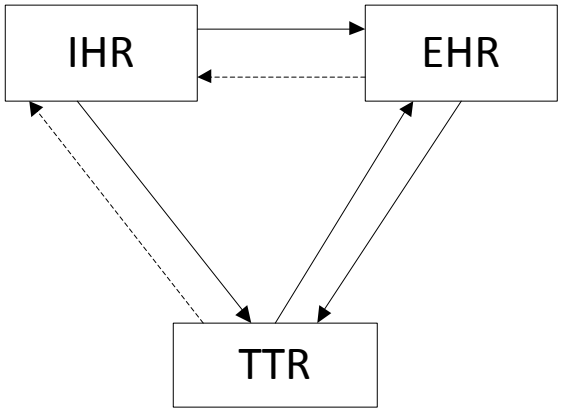


Figure 4.7: Transformations between IHR, EHR and TTR

We represent the table transformations between IHR, TTR and EHR in Figure 4.7. The dotted lines indicate the transformations may not be possible where the solid lines denote the transformations are always feasible.

As IHR is the default table representation supported by WCSs, it is more natural for users to directly issue queries against IHR. However, as we have already seen in Figure 4.2, the implicit TI representation strategy of IHR can cause wrong or misleading results during query processing. Hence, to guarantee the soundness of query processing, an IHR table has to be translated into either a TTR table or an EHR table.

The table transformation tasks could be either automatically inserted by the query processing engine or explicitly specified by users. The former would

correspond to a model where users issue the queries against IHR and the EHR and TTR are only used internally for query processing. However, as not every EHR or TTR table can be transformed back to an equivalent IHR, it is possible that users unexpectedly see the internal table representation in the query result. If users are allowed to perform table transformations explicitly, it implies that the users should also have the ability to process the EHR or TTR tables and the corresponding algebra operators need to be defined. As the first approach is not really transparent to the user, the second approach is more preferable. However, both approaches may be worth considering and are supported by the algebra we present in this thesis. The temporal operators of EHR and TTR will be introduced in Chapter 5.

4.5 Technical constraints of WCSs to support explicit history representation

In section 4.3, we have introduced how the explicit history representation (EHR), in which, each attribute value is attached with an explicit TI, can be supported by wide-column stores (WCSs). Although EHR causes less data redundancy than TTR (tuple-timestamping representation) and reduces the requirement of disk capacity, it is restricted by technical constraints of WCSs, namely:

1. The contents of a single row cannot be separated into two different physical nodes.

2. Each table partition can only store a predefined data volume.

3. No physical "update-in-place" support.

4. Data versions which belong to the same row are stored separately on disk.

When a column is frequently modified, the table partition which contains that column will reach its predefined volume threshold fast. When the volume threshold is met, WCS splits the "parent" partition into two sub-partitions with the same capacity and re-distributes the data in the cluster. During the re-distribution, all the partitions (including both parent partition and

sub-partitions) will be off-line and database access for those partitions will be blocked.

It is possible that at some point the whole partition contains only a single row (as we assume a column in WCS can contain an infinite number of data versions). However, the WCS cannot accept new updates for that row anymore, as the new data version will cause a partition split which may lead the contents of the same row to be stored on different physical nodes. Hence, storing arbitrary number of data versions in an EHR table is impossible.

Different from the relational database systems, a WCS does not physically support "update-in-place". A new data value is appended at the end of the corresponding file (in HBase, files stored in the distributed file system are called "*HFiles*"). Hence, it is usually the case that the contents of one row are separately stored in multiple HFiles. As WCSs have no index support, a sequential access pattern, i.e. scanning all $HFile$s one by one, is the only way to access the whole contents of a tuple. In consequence, it will be very inefficient to obtain a data value when a row contains a lot of data versions.

Due to discussions in the HBase community, a general number of data versions contained for each column is between 1000 to 2000 [Hba]. Note that this number can be increased or decreased when the size of data version varies.

4.6 Summary

In this chapter, we introduced how temporal data could be maintained in WCSs. Although a column in a WCS table can contain multiple data versions, its implicit temporal interval (TI) representation can cause wrong or misleading temporal query results. To avoid that, we proposed two alternative temporal representations, namely, *tuple-timestamping representation* (TTR) and *explicit history representation* (EHR). TTR appends each tuple with an explicit TI where EHR attaches TI into each data version. Comparing these two table representations, EHR stores less data redundancy than TTR.

However, due to the technical constraints of WCSs, it is impossible to store large number of data versions in EHR. Moreover, as a WCS usually has no index support and the contents of a single row may be stored in different files on disk, the performance of data access can be heavily decreased when utilizing EHR.

Chapter 5

Processing temporal data in wide-column stores

As we have already seen in Chapter 4, the *implicit history representation* (IHR) has to be translated into the *explicit history representation* (EHR) or the *tuple-timestamping representation* (TTR) to avoid wrong or misleading results during temporal query processing.

In this chapter, we first describe the architecture of a temporal data warehouse based on WCSs. Then, we introduce a set of temporal operators based on TTR and EHR, respectively. Finally, we describe how these temporal operators can be implemented and compare their performance to indicate which one is more suitable for temporal data processing.

5.1 Architecture of temporal data processing

Figure 5.1 shows the architecture of a temporal data warehouse based on WCSs. The data sources (denoted by superscript I) have the default WCS table representations, i.e. each data version has an implicit TI representation.

The ETL (extract, transform and load) processing monitors and reports the data changes made at the source data, transforms implicit temporal intervals into explicit temporal intervals and loads the results into the DW.

Each extracted table (indicated by superscript T) in the data warehouse has an explicit temporal interval representation and is utilized as a base relation to construct the temporal view definitions (denoted as $E^T(S^T, ..., R^T)$). Different from the temporal DW architecture described by [YW98], we store base relations physically in DW. Each base relation stores a complete data

history and no data will be deleted or overwritten. For performance reason, we cache the results of temporal queries (materialized temporal views) in the data warehouse (denoted as M^T) and the materialized temporal views are modified based on the change of the base relations.

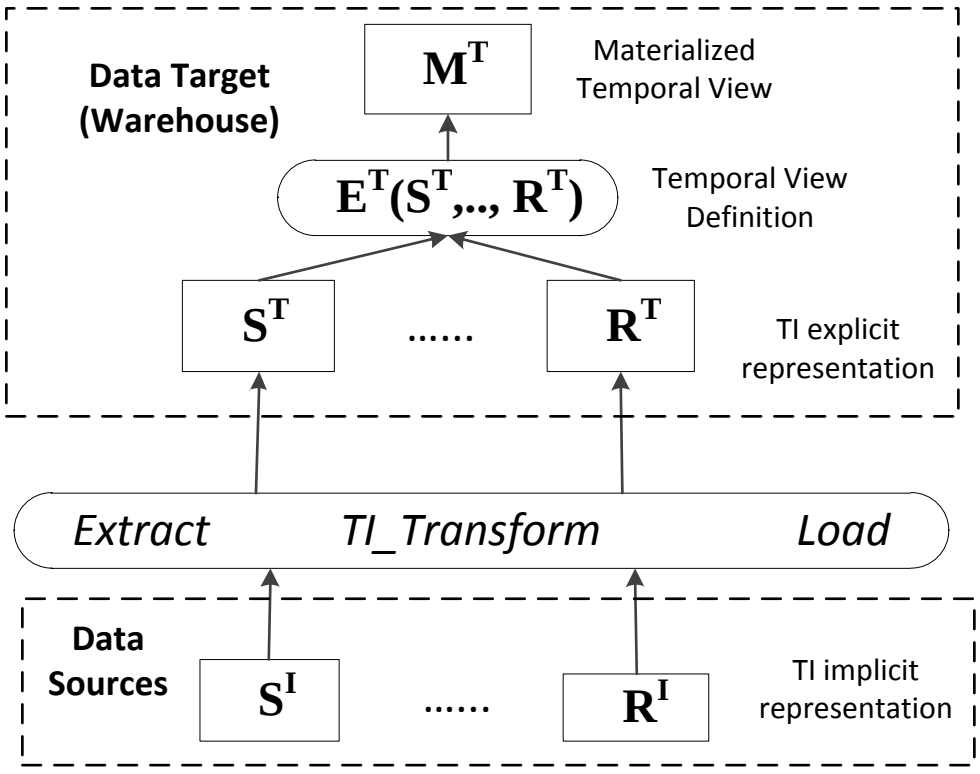


Figure 5.1: Architecture of temporal data warehousing based on WCSs

Notice that, in contrast to the general ETL processing, the functionality of "transform" phase in Figure 5.1 is limited. It only transforms the implicit temporal intervals into the explicit temporal intervals (Temporal interval transformation will be described in Chapter 7) and temporal data analytic is performed in DW.

5.2 TTRO temporal operator model

If temporal relational data is modeled by exploiting *tuple-timestamping*, the temporal relational algebra, which is an extension of the relational algebra,

can be used for temporal data processing. In the context of WCSs, the TTR tables follow the tuple-timestamping model. Intuitively, we can directly utilize the temporal relational algebra for processing TTR tables.

However, different from the general tuple-timestamping tables, to guarantee the unique row key value, each TTR table in the WCSs must integrate the time interval into the row key rather than representing it as separate column(s). Hence, we model the row key in TTR as a pair (srk, TI). srk denotes the row key value derived from its corresponding source (IHR) table and TI indicates the valid time interval which has the form $[Start, End)$.

Moreover, as the row key is mandatory for WCS (TTR) tables, it still must be included in the final results even if it is not indicated in the query. Hence, to satisfy the characteristics of the TTR tables, we extend and customize temporal relational algebra to a temporal operator model called $TTRO$ for the TTR relations. Before presenting the details of TTRO operators, let us first adapt some concepts and definitions from [DBS96] to the TTR context.

Definition 5.2.1 (Value Equivalent). Let r be any TTR table. Two tuples t_1 and t_2 on r are considered as *value equivalent* (written $t_1 \cong t_2$) if and only if all $families : columns$ and $rk.srk$ in both tuples share the same values.

Definition 5.2.2 (Temporal duplicates). Let t_1 and t_2 be two tuples in table S. We say t_1 is a temporal duplicate of t_2 when $t_1 \cong t_2$ and $t_1.TI \cap t_2.TI \neq \emptyset$.

Definition 5.2.3 (Overlap operation). The functionality of the *Overlap operation* is to calculate the intersection of two TIs, namely, $overlap(TI_1, TI_2) = TI_1 \cap TI_2$.

Definition 5.2.4 (Coalesce operation). The functionality of the *Coalesce operation* (denoted by $\boxplus$) is to combine all the value-equivalent tuples of a TTR table together, when their TIs are overlapping or adjacent.

The coalesce operation can be seen in analogy to duplicate elimination in the relational database context. The implementation of $\boxplus$ based on MapReduce is described in Algorithm 8.

The Map function takes the TTR table as input. The intermediate key of output is composed by $(srk + RowResult)$, which causes all tuples which share the same row key and the same column values to be delivered to the

Algorithm 8 Coalesce operation

```
1: Map(RowKey, RowResult)
2: emit(srk + RowResult, TI);
3: Reduce(srk + RowResult, List of TIs)
4: SortedList SL based on TI.Start;
5: for each TI in List do
6:     SL.add(TI);
7: end for
8: TI_n = TI_0; // TI_0 is the first element in SL
9: for each TI_i in SL where SL.length > i > 0 do
10:    if overlap(TI_n, TI_i) ≠ ∅ then
11:        TI_n = [TI_n.Start, TI_i.End);
12:    else
13:        emit(srk + TI_n, RowResult);
14:        TI_n = TI_i;
15:    end if
16: end for
```

same *Reducer*. The intermediate value is TI of each tuple. In the *Reduce* function, we first sort the received TI based on $TI.Start$ in an ascending order. Then we check the relationship between TI_i and TI_j where TI_j is the successor of TI_i. If $overlap(TI_n, TI_i) \neq \emptyset$, we combine these two intervals (lines 12-13). Otherwise, no TIs can be coalesced and a result tuple will be generated. Note that, we can optimize this implementation by adding a *Combine* function between the *Map* and *Reduce* functions.

In the following sections, to simplify the definition of TTRO operators, we utilize $t.TI$, $t.TI.Start$ and $t.TI.End$ to indicate the temporal interval, the start point of TI and the end point of TI for tuple t (Please notice that the temporal interval in TTR is represented as a part of the row key instead of a separate column. Consequently, $t.TI$ is shorthand of $t.rk.TI$). Furthermore, besides giving the formal definition for TTRO operators, we also explain how to use MapReduce to implement them.

5.2.1 Union $\cup^T$

Let r_1 and r_2 be two TTR tables which share the same schema definitions. The union of these two tables is defined as follows:

$$r_1 \cup^T r_2 = \boxplus(r_1 \cup r_2), \text{where } \cup \text{ is the relational union operator.} \quad (5.1)$$

In the definition, we first union the tuples from two tables together and then apply the coalesce operation to combine multiple tuples which are value-equivalent and their TIs are overlapping or adjacent.

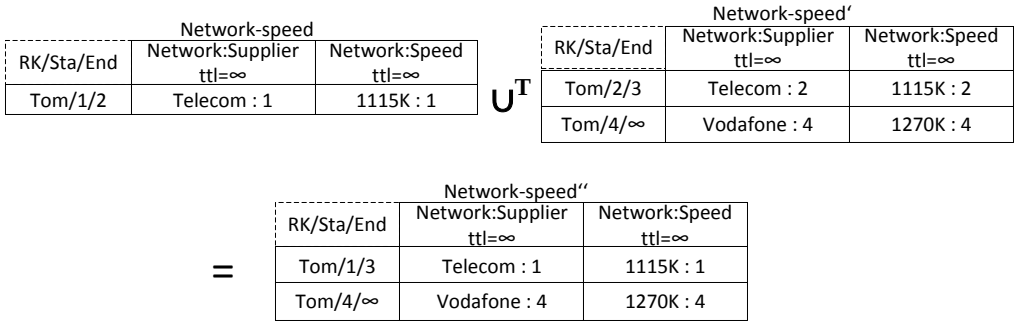


Figure 5.2: Example of TTRO Union operator

Algorithm 9 TTRO Union operation

```
1: Map(RowKey, RowResult)
2: emit(srk + RowResult, TI);
3: Reduce(srk + RowResult, List of TIs)
4: SortedList SL;
5: for each TI in List do
6:     SL.add(TI);
7: end for
8: TI_n = TI_0; // TI_0 is the first element in SL
9: for each TI_i in SL where SL.length > i > 0 do
10:    if overlap(TI_n, TI_i) ≠ ∅ then
11:        TI_n = [TI_n.Start, TI_i.End);
12:    else
13:        emit(srk + TI_n, RowResult);
14:        TI_n = TI_i;
15:    end if
16: end for
```

The implementation of $\cup^T$ based on a MapReduce program is described in algorithm 9.

Figure 5.2 shows an example of $\cup^T$. Tuple "Tom/1/2" and Tuple "Tom/2/3" are value-equivalent and their TIs are adjacent. Hence, they are coalesced into "Tom/1/3".

5.2.2 Difference $-^T$

Let r_1 and r_2 be two TTR tables which share the same schema definitions. The difference of these two tables is defined as follows:

$$r_1 -^T r_2 = \{t|((t \in r_1) \wedge (\rightharpoonup \exists t_2 \in r_2|(t \cong t_2) \wedge (overlap(t, t_2) \neq \emptyset)))\vee \\ (\exists t_1 \in r_1, \exists t_2 \in r_2|(t_1 \cong t_2) \wedge (t \cong t_2) \wedge (t.TI \in \{(t_1.TI - overlap(t_1, t_2))\})) \\ \wedge (\{(t_1.TI - overlap(t_1, t_2))\} \neq \emptyset) \wedge (overlap(t_1, t_2) \neq \emptyset)\}. \tag{5.2}$$

In the difference definition, tuples in r_1 will be directly emitted, when there does not exist any tuples in r_2 in which they are value-equivalent and their TIs have overlaps (line 1). Otherwise, TIs in r_1 need to be modified.

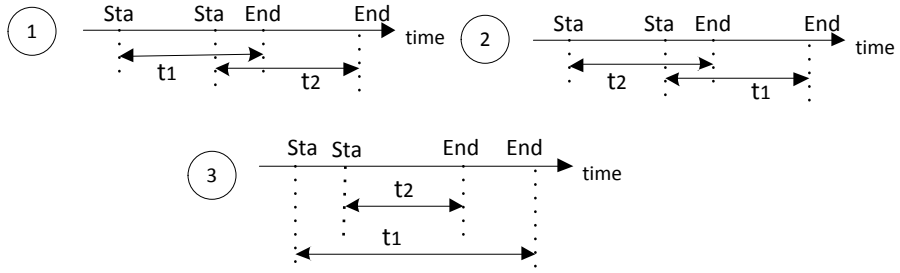


Figure 5.3: Various temporal relationships between t_1 and t_2

Figure 5.3 displays 3 possible temporal relationships between t_1 and t_2 to denote the values of $(t_1.TI - overlap(t_1, t_2))$, namely:

1. $[t_1.TI.Sta, t_2.TI.Sta)$;
2. $[t_2.TI.End, t_1.TI.End)$;
3. $[t_1.TI.Sta, t_2.TI.Sta), [t_2.TI.End, t_1.TI.End)$.

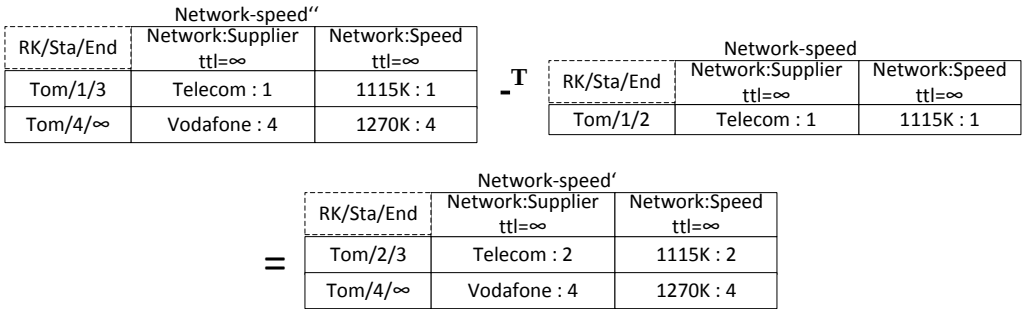


Figure 5.4: Example of TTRO Difference operator

Please notice that, for the third situation, one tuple in r_1 will be split into two tuples.

An example of $-^T$ is shown in Figure 5.4. Tuple "Tom/1/3" and Tuple "Tom/1/2" are value-equivalent and their TIs are overlapping. Hence, the TI of "Tom/1/3" is changed into $[2, 3)$. For tuple Tom/4/∞, no value-equivalent tuples can be found in r_2.

Algorithm 10 TTRO Difference operation

```
1: Map(RowKey, RowResult)
2: Meta = TI + tag;
3: emit(srk + RowResult, Meta);
4: Reduce(srk + RowResult, List of Metas)
5: TIList LTI_1, LTI_2; //TIs from r_1 and r_2
6: for each TI_1 in LTI_1 do
7:     TI_t = TI_1;
8:     for each TI_2 in LTI_2 do
9:         TI_t = TI_t − TI_2;
10:    end for
11:    if TI_t ≠ ∅ then
12:        emit(srk + TI_t, RowResult);
13:    end if
14: end for
```

The implementation of $-^T$ is depicted in Algorithm 10. To implement the $-^T$ operator, the Map function adds a tag to each intermediate key-value

pair to denote whether the value belongs to r_1 or r_2 (line 2). All the tuples which are value-equivalent but with different TIs will be delivered to the same *Reduce* node. In the *Reduce* function, two array-lists (LTI_1 and LTI_2) are created to store the TIs from table r_1 and r_2 (line 5), respectively. For each TI in LTI_1, we subtract it with each TI in LTI_2. When result TI is not empty, we emit the result tuple.

5.2.3 Intersection $\cap^T$

Let r_1 and r_2 be two TTR tables which share the same schema definitions and the definition of $\cap^T$ is given as follows:

$$r_1 \cap^T r_2 = \{t|(\exists t_1 \in r_1, \exists t_2 \in r_2|(t \cong t_2 \cong t_1) \wedge (t.TI = overlap(t_1, t_2) \neq \emptyset))) \tag{5.3}$$

We can also derive the definition of $\cap^T$ from $-^T$, such as:

$$r_1 \cap^T r_2 = r_1 -^T (r_1 -^T r_2). \tag{5.4}$$

Algorithm 11 TTRO Intersection operation

```
1:  Map(RowKey, RowResult)
2:  Meta = TI + tag;
3:  emit(srk + RowResult, Meta);
4:  Reduce(srk + RowResult, List of Metas)
5:  TIList LTI_1, LTI_2; //TIs from r_1 and r_2
6:  for each TI_1 in LTI_1 do
7:      TI_t = TI_1;
8:      for each TI_2 in LTI_2 do
9:          TI_t = TI_t ∩ TI_2;
10:         if TI_t ≠ ∅ then
11:             emit(srk + TI_t, RowResult);
12:         end if
13:     end for
14: end for
```

The implementation of $\cap^T$ is described in Algorithm 11. The functionality of *Map* is identical to Algorithm 10. In *Reduce*, we detect whether TIs in r_1

are overlapping with TIs in t_2 (lines 8-13). When such a TI overlap can be found, we emit the result tuple.

5.2.4 Filter σ_p^T

Let r_1 be a TTR table. Let p denote a selection condition over attributes of r_1. For a better explanation, we classify predicate p into four different categories:

1. No predicate: $p = empty$;
2. Non-temporal predicate: $p = a\theta b$, where $\theta \in \{\leq, <, \geq, >, =, \neq\}$, a and b can be atomic value constants, $rk.srk$, $Column.Value$;
3. Temporal predicate: $p = a\theta b$, where $\theta \in \{\leq, <, \geq, >, =, \neq\}$, a and b can be temporal constants, $TI.Start$, $TI.End$;
4. Predicate with logical connectives: $p\theta p$, where $\theta \in \{\wedge, \vee\}$.

The definition of σ_p^T is given as follows:

$$\sigma_p^T(r_1) = \{t | (t \in r_1) \wedge (p(t) = true)\}. \tag{5.5}$$

The corresponding example of filter operation can be found in Figure 4.4. The implementation of σ_p^T based on MapReduce is described in Algorithm 12.

Algorithm 12 TTRO Filter operation

```
1: Map(RowKey, RowResult)
2: if RowResult makes p true then
3:     emit( RowKey, RowResult);
4: end if
```

Please note that temporal comparison operators (syntactic sugar) such as *Allen's interval operators* [All83] or *period predicates* supported by SQL 2011 [KM12] can be easily translated into temporal conditions (line 3) with logical connectives (line 4). For example, the *Contains(TS, TI)* predicate detects whether a time instant TS is included in a temporal interval TI. This can be specified as a temporal predicate $TI.Start \leq TS < TI.End$. To denote the current time in filter predicate, symbol NOW is used. The value of NOW automatically increases when time elapses.

5.2.5 Projection π_A^T

In the temporal relational algebra, a project operation will return the columns which are indicated by the set of desired projection attributes A. Moreover, TI will be calculated in the final result even if it is not explicitly specified in the projection attributes.

In the TTR context, to guarantee that the output of projection is still consistent with the data model of WCSs (TTR tables), the row key must be "implicitly" included in each tuple. We say "implicitly" because the row key must not be specified in the projection attributes. The definition of π_A^T is given as follows:

$$\begin{aligned}\pi_A^T(r_1) = \{t | \exists t_1 \in r_1 | (t.srk = t_1.srk) \wedge (t[A_1] = t_1[A_1]) \wedge ... \wedge \\ (t[A_n] = t_1[A_n])\}. \\ \text{where each } A_i \text{ is the name of a column or a column-family.}\end{aligned} \tag{5.6}$$

The implementations of π_A^T is described as follows:

Algorithm 13 TTRO Projection operation

```
1: Map(RowKey, RowResult)
2: Result = RowResult[A];
3: emit(RowKey, Result);
```

To implement π_A^T, only a Map function is needed.

5.2.6 Cartesian product $\boxtimes^T$

Let r_1 and r_2 be two TTR tables. The Cartesian product of these tables is defined as follows.

$$\begin{aligned}r_1 \boxtimes^T r_2 = \{t | \exists t_1 \in r_1, \exists t_2 \in r_2 | (t[Atrr(R_1) - rk] = t_1[Atrr(R_1) - rk]) \wedge \\ (t[Atrr(R_2) - rk] = t_2[Atrr(R_2) - rk]) \wedge (t.srk = \\ concat(t_1.srk, t_2.srk)) \wedge (t.TI = (overlap(t, t_2)) \wedge (t.TI \neq \emptyset))\}.\end{aligned} \tag{5.7}$$

As WCSs do not support composite keys (one table can only contain a single row key column), we define a *concat* function to concatenate the *srk*s

Network-speed

RK/Sta/ End	Network:Supplier ttl=∞	Network:Speed ttl=∞
Tom/1/3	Telecom : 1	1115K : 1
Tom/4/∞	Vodafone : 4	1270K : 4

$\boxtimes^T$

Manager-info

RK/Sta/ End	Company:Name ttl=∞
Jim/2/3	1&1 : 2
Joe/7/9	Vodafone : 7

=

Network-info

RK/Sta/ End	Network:Supplier ttl=∞	Network:Speed ttl=∞	Company:Name ttl=∞
Tom/Jim/ 2/3	Telecom : 2	1115K : 2	1&1 : 2
Tom/Joe/ 7/9	Vodafone : 7	1270K : 7	Vodafone : 7

Figure 5.5: Example of TTRO Cartesian operator

from both tuples (line 3). Moreover, both tuples have to be temporally valid during the same time period (line 3).

Figure 5.5 shows an example of $\boxtimes^T$. The result table contains only one tuple, i.e. "Tom/Green/7/9" which is derived from "Tom/4/∞" (in "Network-speed" table) and "Green/7/9" (in "Manager-info").

The implementation of $r_1 \boxtimes^T r_2$ is not trivial in MapReduce. In general, three approaches are possible:

1. Using DistributedCache mechanism supported by MapReduce

Algorithm 14 TTRO Cartesian operation

```
1: DistributedCache.addCacheFile(URI, JobConf);
2: Configure(JobConf)
3: File f = new File(Path);
4: Map(RowKey, RowResult)
5: for each tuple t_1 in f do
6:     TI_n = overlap(RowKey.TI, t_1.TI);
7:     if TI_n is not empty then
8:         srk_n = RowKey.srk + t_1.srk;
9:         Result = RowResult + t_1.Value;
10:        emit(srk_n + TI_n, Result);
11:    end if
12: end for
```

DistributedCache is a facility provided by the MapReduce framework to cache files (text, archives, jars etc.) needed by applications. Applications specify the files via URLs and the framework will copy the necessary files to a slave node before any tasks for the job are executed on that node. Utilizing this facility, we can store a small table in the distributed cache. The implementation is described in Algorithm 14.

Line 1 illustrates how to add a file to the distributed cache. $Configure$ (line 2) is a method which is called before the Map function is executed. It initializes a *File* object f (which references table r_1) and copies r_1 to every work node. The Map function takes every tuple t_2 in r_2 as input and checks if $t_2.TI$ has a TI overlapping with $r_1.TI$. Please note that the maximum file size which can be stored by *DistributedCache* is 10G.

2. Duplicating a table by using the HBase API

In contrast to the first approach, we can also duplicate the small table by utilizing the HBase API, namely, calling the "Scan" operation inside the Map function. This strategy avoids the volume limitation of *DistributedCache*. Algorithm 15 describes how this approach can be implemented in the MapReduce framework.

Algorithm 15 TTRO Cartesian operation

```
1: Map(RowKey, RowResult)
2: Result re = Scan(r1);
3: for each tuple t1 in re do
4:     TI_n = overlap(t1.TI, RowKey.TI);
5:     if TI_n is not empty then
6:         srk_n = t1.srk + RowKey.srk;
7:         res_n = t1.Value + RowResult;
8:         emit(TI_n + srk_n, res_n);
9:     end if
10: end for
```

Suppose we have two tables r_1 and r_2 where $size(r_1) = m$, $size(r_2) = n$ and $m < n$. We will then broadcast table r_1 to each Map node. To implement $r_1 \boxtimes^T r_2$, for every tuple t_2 in r_2, we scan r_1 (line 2) and check whether t_2 has a TI overlapping in r_1 (lines 3-4). When such tuples are found, we concatenate the row keys from those two tuples and generate the result tuple (lines 5-10).

3. Defining a customized file input format

The third approach is to define a customized file input format, e.g. *CartesianInputFormat*. This input format takes two tables as input. For every split of r_1, we append the complete table content of r_2 with it. At last, a user-defined RecordReader, .e.g. *CartesianRecordReader*, denotes how input key-value pairs for the Map function can be built.

Notice that, the first and the third approaches are only suitable for pure MapReduce environment and hence force us to dump the content of two WCS tables in the distributed file system e.g. the Hadoop File System. After data processing is finished, we have to insert the result table back into WCSs. The second approach can avoid both table dumping and table insertion. Hence, we prefer the second one.

5.2.7 Theta-Join $\bowtie_p^T$

The Theta-join operation $\bowtie_p^T$ can be considered as a Cartesian product $\boxtimes^T$ of two TTR tables followed by a filter operation σ_p^T. The definition of $\bowtie_p^T$ is given as follows:

$$r_1 \bowtie_p^T r_2 = \sigma_p^T(r_1 \boxtimes^T r_2). \tag{5.8}$$

As p may contain non-equal predicates, $\bowtie_p^T$ can only be implemented by duplicating the "small" table. The corresponding algorithm can be found in Algorithm 15. Note that, when specifying the temporal natural join operation, a temporal project π_A^T is needed to discard the common attributes. When implementing the temporal natural join, we can either perform it on the Map side or on the *Reduce* side. Algorithm 16 describes how to implement the *Temporal Natural Join* by utilizing the *Reduce* function. The Map function utilizes the join attributes as the intermediate output key and attaches each tuple with a tag to denote to which table it belongs. The *nested-loop* mechanism is implemented inside the *Reduce* function. Besides checking the join predicates, we also explicitly evaluate if two tuples are valid during the same period of time.

Algorithm 16 TTRO Natural Join

```
1: Map(RowKey, RowResult)
2: rk_n = RowResult[Atrri(p)];
3: val_n = RowResult + tag;
4: emit(rk_n, val_n);
5: Reduce(rk_n, List of val_n)
6: List l_1, l_2;
7: for each tuple t in List of val_n do
8:     if tag==r_1 then
9:         l_1.add(t);
10:    else
11:        l_2.add(t);
12:    end if
13: end for
14: for each element t_1 in l_1 do
15:    for each element t_2 in l_2 do
16:        TI_n = overlap(t1.TI, t_2.TI);
17:        if TI_n is not empty and p(t_1, t_2) == true then
18:            srk_n = t_1.srk + t_2.srk;
19:            res_n = t_1.Value + RowResult;
20:            emit(TI_n + srk_n, res_n);
21:        end if
22:    end for
23: end for
```

5.2.8 Temporal aggregation $f_{A'}(\gamma^{T}_{(A,TG,W)})$

In the context of data warehousing and OLAP (online analytical processing), grouping tuples based on different categories and then applying aggregation functions are the most common forms of data processing. In SQL, the aggregation result is generated by the optional *Group-by* combined with aggregation functions, such as SUM, AVG, MIN, MAX and $Count$.

In the non-temporal context, tuples in a table can be grouped based on one or more than one non-temporal attributes. In the temporal context, tuples in a temporal table can be grouped based on:

1. non-temporal attributes;

2. temporal information (temporal instance or temporal interval); and

3. both (i.e. the combination of 1 and 2).

When grouping a temporal table based on temporal information (situation 2), several kinds of temporal aggregation can be defined:

1. *Instantaneous temporal aggregation* (ITA) [KS95] in which the timeline is partitioned based on the finest time granularity supported by the temporal database and the aggregation value at each time t is calculated using all the tuples which are valid at time t. Consecutive time instants which share the same aggregation value will be coalesced together.

2. *Moving-window or cumulative temporal aggregation* (CTA) [LO09, JG09] in which the timeline is partitioned based on the finest time granularity supported by the temporal database and the aggregation value at each time t is calculated using all the tuples which are valid during $[t - m, t]$ where m denotes the length of the moving window. Consecutive time instants which share the same aggregation value will be coalesced together.

In contrast to non-temporal aggregation, each result of the temporal aggregation should be associated with a temporal validity interval (TI). Although the aggregation functions for both ITA and CTA are evaluated at a single time point, the main difference between them is that ITA groups the tuples based on each single time point t where CTA groups tuples which are valid before each single time point t (during $[t - m, t]$).

We use $f_{A'}(\gamma^T_{(A,TG,W)})(r)$ to denote the temporal aggregation based on table r:

- $\gamma^T_{(A,TG,W)}$ is a temporal group-by operator where
  - A denotes the non-temporal grouped attributes;
  - TG indicates the finest time granularity of r and its value can be second, minute, hour, day and so on.
  - W can be assigned by non-negative integer.
- $f_{A'}$ represents the aggregation functions, such as SUM, AVG, MIN, MAX and $Count$, which are applied to attribute A'.

Network

RK/Sta/End	Network:Supplier ttl=∞	Network:Speed ttl=∞
Tom/1/5	Telecom : 1	1000K : 1
Jim/2/8	Telecom : 2	900K : 2

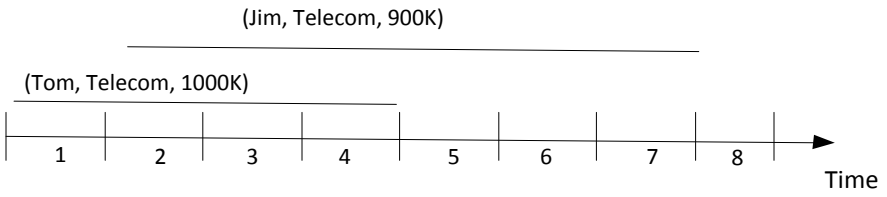


Figure 5.6: Tabular and graphical representations for Network

To specify non-temporal aggregation, TG and W are not set (or set to *null*). To specify different types of temporal aggregation,

1. we set TG to the finest time granularity of r and W to 0 to indicate ITA.

2. we set TG to the finest time granularity of r and W to positive integer to denote CTA.

For example, suppose we have the following queries and "Network" table in Figure 5.6. The finest granularity of "Network" is *day*. For simplicity, we use "Speed" and "Supplier" to represent "Network:Speed" and "Network:Supplier", respectively.

• **Query 1**: What is the average Internet speed of different Internet suppliers for each day?

This query is ITA and specified as $AVG_{Speed}(\gamma^{T}_{(Supplier,day,0)})(Network)$.

• **Query 2**: What is the average Internet speed of different Internet suppliers last two days?

This query is CTA and specified as $AVG_{Speed}(\gamma^{T}_{(Supplier,day,2)})(Network)$.

Before giving the formal definitions of ITA and CTA, we extend the timeslice operator S_t [SAea94] to $S_{(t,w)}$. $S_{(t,w)}(R^T)$ returns a temporal point-based relation based on the value of w.

Definition 5.2.5 (Extended timeslice operator $S_{(t,W)}$). Let R^T be a TTR table with schema (A_{non}, TI) where A_{non} and TI indicate the non-temporal attributes and temporal interval of R^T, respectively. The schema of output of $S_{(t,W)}(R^T)$ is represented as (A_{non}, TP) where TP denotes time point.

$$\begin{aligned} S_{(t,w)}(R^T) =& \{r | \exists r_1 \in R^T | (r_1.TI \cap [t-w, t] \neq \emptyset) \wedge \\ & (r[A_{non}] = r_1[A_{non}]) \wedge (r[TP] = t)\}. \end{aligned} \tag{5.9}$$

When w is 0, $S_{(t,w)}(R^T)$ returns a point-based relation which contains tuples that are valid at time t. When w is assigned by a positive integer, tuples which are valid at t are calculated based on tuples that are valid during $[t-w, t]$.

We give the definition of ITA and CTA as follows:

$$f_{A'}(\gamma^T_{(A,TG,W)}(R^T) = FOLD(f_{A'}(\gamma_{(A,TG)}(\bigcup_{i=1}^{n}(S_{(t_i,w)}(R^T)))), \text{where } t_i \in R^T.TI, \\ \gamma \text{ is non-temporal group-by operator and } \cup \text{ is non-temporal union operator.} \tag{5.10}$$

In the previous definition, we first calculate aggregation groups based on different time points and non-temporal attributes $\gamma_{(A,TG)}(\bigcup_{i=1}^{n}(S_{(t_i,w)}(R^T)$. Then, aggregation function f is applied to attribute A'. Finally, $FOLD$ operator is used to consolidate the consecutive temporal instants which share the same values. The formal definition of $FOLD$ can be found in [LM97].

GP/Sta/End	Res:AVG ttl=∞
Telecom/1/2	1000K : 1
Telecom/2/5	1450K : 2
Telecom/5/8	900K : 5

a

GP/Sta/End	Res:AVG ttl=∞
Telecom/1/2	1000K : 1
Telecom/2/7	1450K : 2
Telecom/7/8	900K : 7

b

Figure 5.7: Results for different sorts of temporal aggregations

In Figure 5.7, tables a and b show the results of queries 1 and 2. We use GP to denote the values of non-temporal grouped attributes. For Query 1, only one tuple "(Tom, Telecom, 1000K)" is valid at time point 1. Hence, the aggregation result is "(1000K, [1, 2))". At time point 2, two tuples "(Tom,

Telecom, 1000K)" and "(Jim, Telecom, 900K)" are valid and the aggregation result will be "(1450K, [2, 3))". Time points 3-4 have the same aggregation value as time point 2. Consequently, their temporal aggregation results are coalesced into "(1450K, [2, 5))". For time points 5-7, only one tuple "(Jim, Telecom, 900K)" is valid. For Query 2, the aggregation value of each time t is calculated based on tuples that are valid during $[t-2, t]$. In consequence, time points 5 and 6 have same aggregation value as time point 4.

To implement ITA and CTA based on MapReduce, two MapReduce jobs are needed. The first one is to group and calculate the aggregation value based on each distinct non-temporal grouped attribute and time instant. The second one is to coalesce the adjacent time-points which share the same aggregation value with the same non-temporal attribute.

5.3 CTO temporal operator model

As described in Chapter 4, the EHR representation uses the *attribute-timestamping model*. Each column in an EHR table maintains multiple data versions associated with explicit temporal intervals (TIs). To process an attribute-timestamped table, [TG89, Tan97] extend the relational algebra to so-called *historical relational algebra*.

The historical relational algebra can be seen as a mixture of non-temporal relational algebra and temporal relational algebra with a set of temporal table and temporal tuple reconstruction operators. For example, they reuse the non-temporal relational operators, such as *Intersection*, *Union*, *Difference* and *Cartesian product* as the basic operations without considering the temporal properties kept in each tuple. Moreover, the filter operator will select a complete row even if only a subset of data versions of that row satisfy the filter predicates. Like the temporal-project operator utilized to process tuple-timestamped tables, the project operator in the historical relational algebra will only output the columns which are specified as projection attributes.

To process temporal data in the EHR context, we propose a novel temporal operator model called *CTO*. The *CTO* model is defined based on the following goals:

1. Each *CTO* operator can be directly applied to EHR tables without first changing the table representation.

2. The temporal operator should generate the *correct* temporal tuples (data versions). For example, when a data version satisfies the selection predicate, to generate a complete row, the TI of that data version will be utilized as a selection criterion to fetch the data versions from the other columns which have a temporal overlap with it.

3. The class of EHR tables is closed under the CTO model, namely, the output of each CTO operator must still be an EHR table.

To define the semantics of each CTO operator, we have two options:

- The first one is to define each operator directly.
- The second one is to utilize the table transformation operations T_{ET} and T_{TE} (described in Chapter 4) together with the $TTRO$ operators. We call this approach "*indirect definition*".

In general, the first approach is more complicated and difficult to understand (compared to the second one), as the relationships between data versions need to be specified in the definition. For better comparison of these two approaches, we will give both *direct* and *indirect definitions* for the Union operator $\cup^T$ as examples.

Moreover, for each CTO operator, we also indicate how to implement it by utilizing the MapReduce framework. Please note that, as the granularity of DML (data manipulation language) in WCS is based on version, a row-insertion is decomposed of a set of version-insertions.

5.3.1 Union $\cup^c$

Let r_1 and r_2 be two EHR tables which share the same schema definitions. The direct and indirect definitions are given as follows.
Direct definition:

$$\begin{aligned} r_1 \cup^c r_2 = \{t | & (t \in r_1) \wedge (\forall t_2 \in r_2, t[RK] \neq t_2[RK]) \vee (t \in r_2) \wedge \\ & (\forall t_1 \in r_1, t[RK] \neq t_1[RK]) \vee (\exists t_1 \in r_1, \exists t_2 \in r_2 | \\ & (t[RK] = t_1[RK] = t_2[RK]) \wedge (\forall CF_m \in Attr(R_1) | \\ & \forall Col_{mn} \in CF_m | (t[CF_m.Col_{mn}] = \boxplus(t_1[CF_m.Col_{mn}] \cup t_2[CF_m.Col_{mn}]))\}, \\ & \text{where } \cup \text{ is a set union operator and } \boxplus \text{ is the coalesce operation.} \end{aligned} \tag{5.11}$$

Tuples from both tables will be included in the final results if there is no tuple in the other table which has the same row key. Otherwise, we will first union the data versions of equally named columns and then apply the coalesce operation to combine value-equivalent data versions in which the TIs are overlapping or adjacent.
Indirect definition:

$$r_1 \cup^c r_2 = T_{TE}(T_{ET}(r_1) \cup^T T_{ET}(r_2)). \tag{5.12}$$

To achieve the same operational semantics described in the equation 5.11, two EHR tables will be first transformed into their corresponding TTR tables via T_{ET} operation. Then, $TTRO$ Union operator $\cup^T$ is applied to these two relations. Finally, the result TTR table will be transformed back to its correlated EHR table. Notice that, the semantics of coalesce operation is implemented inside T_{TE}.

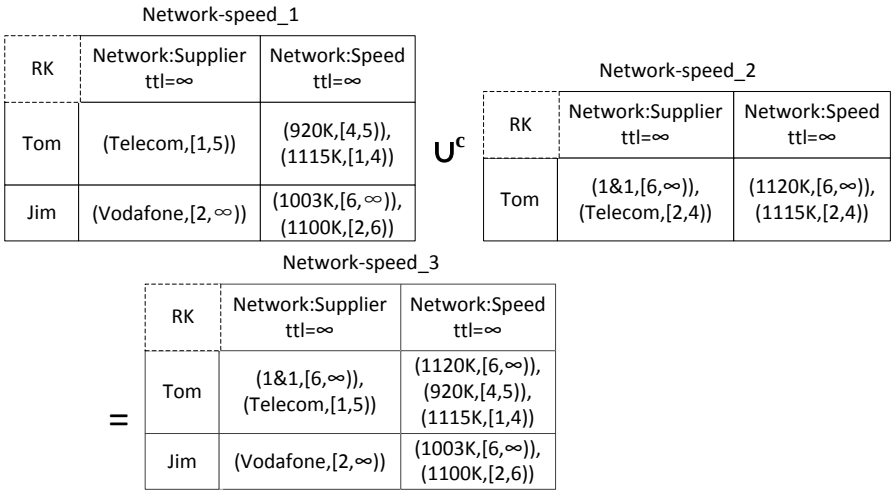


Figure 5.8: Example of CTO Union operator

As illustrated by the previous two definitions, we can notice that the indirect definition uses less space and is easier to understand. In the following sections, we therefore utilize the indirect way, namely, $TTRO$ operators together with the table transformation operations (T_{ET} and T_{TE}) to define the semantics of CTO operators. Note that this is only for definitional purposes. The implementations of CTO operators do not perform transformations to TTR and back to EHR.

The corresponding example of $\cup^c$ is shown in Figure 5.8. When tuples from two tables share no key values, they will be directly placed in the final result. Otherwise, we first union the data versions from two columns together and later apply the coalesce operation. In Figure 5.8, row "Jim" in table "Network-speed_1" cannot find any tuple in "Network-speed_2" with the same row key. Hence, it is directly included into the final result. Row key "Tom" exists in both tables. To generate the desired data result, we first union each column together. Then, for every column, we will consolidate data versions which are value-equivalent and their TIs are overlapping or adjacent. For example, data version "(Telecom, [1, 5))" in "Network-speed_1" and data version "(Telecom, [2, 4))" in "Network-speed_2" are combined into "(Telecom, [1, 5))" in "Network-speed_3".

Algorithm 17 CTO Union operation

```
1: Map(RowKey, RowResult)
2: for each data version D in RowResult do
3:     rk_n = RowKey + Col + CF + D_v;
4:     emit(rk_n, D_TI);
5: end for
6: Reduce(rk_n, List of D_TI)
7: SortedList sl;
8: for each TI in List of D_TI do
9:     sl.add(TI);
10: end for
11: TI temp;
12: temp = sl.get(0);
13: for each element TI_i in sl, where 0 ≤ i < sl.size() do
14:     if overlap(TI_i, temp) ≠ ∅ or temp.End == TI_i.Start then
15:         temp = temp ∪ TI;
16:     else
17:         res_n = rk_n.Col + rk_n.CF + rk_n.D_v + temp;
18:         emit(rk_n.RowKey, res);
19:         temp = TI_i;
20:     end if
21: end for
```

The implementation of U^c based on the MapReduce programming model is described as above.

The intermediate key of the *Map* function is formed as row key, column-family name, column name and the value of each data version. This strategy guarantees that tuples from two tables which share the same row key and same column value will be sent to the same Reducer. In the *Reduce* function, the TI of value-equivalent data versions will be consolidated together. We first detect if the two data versions have TI overlapping or their TIs are adjacent (line 14). In that case, two TIs will be combined. Otherwise, a result data version will be emitted (lines 17-19).

5.3.2 Difference $-^c$

Let r_1 and r_2 be two EHR tables which share the same schema definitions. The difference of these two tables is defined as follows:

$$r_1 -^c r_2 = T_{TE}(T_{ET}(r_1) -^T T_{ET}(r_2)). \tag{5.13}$$

The equation above indicates that tuples in r_1 do not match tuples in r_2 with the same row key can be directly placed into the final result. Otherwise, for each column in r_1, the TIs of data versions in which they are value-equivalent and have TI overlapping with data versions in r_2 will be modified.

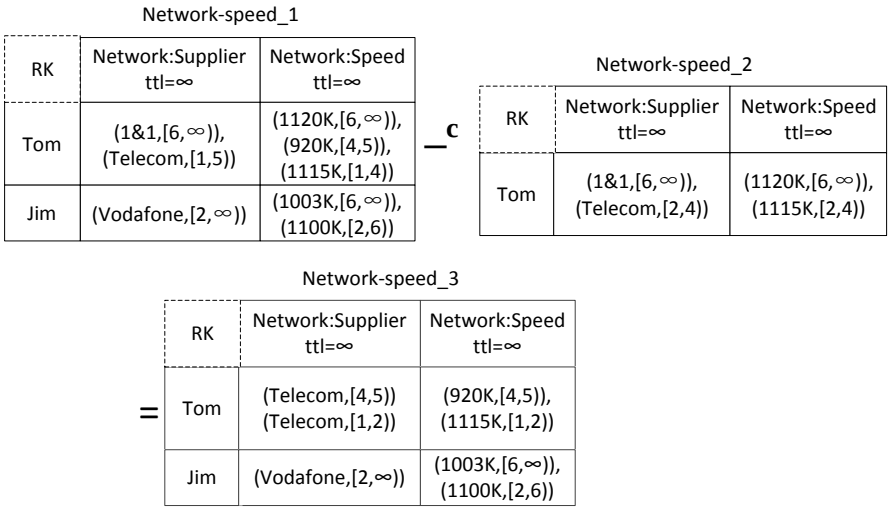


Figure 5.9: Example of CTO Difference operator

Figure 5.9 shows an example. Row "Jim" in "Network-speed_1" does not match any tuples in "Network-speed_2" with the same row key. In conse-

quence, it can be directly included in the output result. However, for tuple "Tom" t_1 in "Network-speed_1" and "Tom" t_2 in "Network-speed_2", we first form two collections of TIs (TI_{s1} and TI_{s2}) based on these two tuples. For tuples t_1 and t_2, we get TI_{s1}=[1,4), [4,5), [5,6), [6,∞) and TI_{s2}=[2,4), [4,∞), respectively. Then, we split t_1 based on TI_{s1} and t_2 based on TI_{s2}. This tuple split operation is necessary as we need to guarantee that data versions are discarded only when both t_1 and t_2 share the same tuple values during the same period time. As each tuple now has a TTR representation, we can directly calculate $r_1 -^T r_2$. Finally, we transform the TTR representation back to EHR representation.

Algorithm 18 CTO Difference operation

```
1: Map(RowKey, RowResult)
2: res = RowResult + tag;
3: emit(RowKey,res);
4: Reduce(RowKey, List of res)
5: SortedList sl_1, sl_2;
6: put TS in sl_1 where res.tag == r_1;
7: put TS in sl_2 where res.tag == r_2;
8: TI-Set s_1, s_2;
9: derive s_1 from sl_1;
10: derive s_2 from sl_2;
11: List l_1, l_2, l_3;
12: derive tuples from t_1 based on s_1 and store in l_1;
13: derive tuples from t_2 based on s_2 and store in l_2;
14: for every element t_1 in l_1 do
15:     for every element t_2 in l_2 do
16:         if t_1 ≅ t_2 and overlap(t_1, t_2) ≠ ∅ then
17:             t_1.TI = t_1.TI − t_2.TI;
18:         end if
19:     end for
20:     l_3.add(t_1.TI + t_1.Value);
21: end for
22: HashMap < Col + CF + Val, SortedList < TI >> hms;
23: put each element in l_3 to hms based on its column value;
24: Combine TIs in hms and insert into result table;
```

Algorithm 18 describes the implementation of $-^c$ which is a combination

of Algorithm 6, Algorithm 10 and Algorithm 7. In the Reduce phase, we first transform the EHR tuple into a set of TTR tuples (lines 5-13). Then we implement the difference operation (lines 14-21). Finally, we create a Hashmap to imitate the shuffle phase and combine TIs which are overlapping or adjacent. Note that, although this implementation puts all these operations in a single MapReduce procedure, it will need large memory size when the number of data versions is large.

5.3.3 Intersection $\cap^c$

Let r_1 and r_2 be two EHR tables which share the same schema definitions. The definition of $\cap^c$ is derived from $-^c$:

$$r_1 \cap^c r_2 = r_1 -^c (r_1 -^c r_2). \tag{5.14}$$

In analogy to the $-^c$ operation, a data version D_1 in r_1 will only be discarded if and only if:

- There exists a data version D_2 in the same column in r_2 in which $D_1 \cong D_2$ and $overlap(D_1, D_2) \neq \emptyset$;
- For all D_i in r_1 where $(D_i.rk = D_1.rk) \wedge (overlap(D_i, D_1) \neq \emptyset)$, we can guarantee that for all $D_j \in r_2$ where $(D_j.rk = D_2.rk)$, the equation $(D_j \cong D_i) \wedge (overlap(D_i, D_j) \neq \emptyset)$ will return true.

The implementation of $\cap^c$ based on MapReduce programming model is almost the same as $-^c$. Instead of calculating $t_1.TI = t_1.TI - t_2.TI$ (line 17 in Algorithm 18), $t_1.TI = t_1.TI \cap t_2.TI$ is performed.

5.3.4 Projection π^c_A

Let r_1 be an EHR table. $\pi^c_A(r_1)$ will output the desired column values which are specified in A and TI are also included in the result data versions.

$$\pi^c_A(r_1) = T_{TE}(\pi^T_A((T_{ET}(r_1))). \tag{5.15}$$

Different from the $TTRO$ project operator π^T_A:

1. Each column in the output of π^c_A contains multiple data versions.

2. TI is integrated into each data version instead of into the row key.

The implementation of π_A^T is described as follows. As already described in Section 2, the granularity of DML (data manipulation language) in WCS is based on version. In consequence, a row-insertion is decomposed into a set of version-insertions (line 2-4).

Algorithm 19 CTO Projection operation

```
1: Map(RowKey, RowResult)
2: for every data version D in A do
3:     emit(RowKey, D);
4: end for
```

5.3.5 Filter σ_p^c

Let r_1 be an EHR table. Different from the filter operator proposed in [Tan97], $\sigma_p^c(r_1)$ only outputs the data versions D_f which satisfy the selection predicate and the corresponding data versions D_c from the other columns in which TI of D_c is adjusted based on D_f. The definition of $\sigma_p^c(r_1)$ is given as follows:

$$\sigma_p^c(r_1) = T_{TE}(\sigma_{p'}^T(T_{ET}(r_1))). \tag{5.16}$$

In the definition, the temporal comparisons in predicate p which are applied to EHR table has to be translated into p' in the TTR context. The reason is TI in the TTR table is integrated into the row key rather than into each data version. Note that, this predicate transformation is merely utilized to define the semantics of σ_p^c. The real CTO filter implementation does not perform this transformation. The implementation of σ_p^c based on MapReduce is described below.

The corresponding example is shown in Figure 4.3. To select the Internet supplier whose speed has at any time been faster than 1000K, the data versions "(1115K, [1, 2))", "(1855K, [2, 3))" and "(1270K, [4, ∞))" satisfy the selection predicate where "(920K, [3, 4))" does not. To generate a complete satisfied row, data versions "[Vodafone, [4, ∞))" and "[Telecom, [1, 3))" are selected from the "Network:Supplier" column.

Algorithm 20 CTO Filter operation

```
1: Map(RowKey, RowResult)
2: for every data version D in RowResult do
3:     if D satisfies p then
4:         select D_c from the other columns as D_c.TI = D_c.TI ∩ D.TI ≠ ∅
5:         emit(RowKey, D);
6:         emit(RowKey, D_c);
7:     end if
8: end for
```

5.3.6 Cartesian product $\boxtimes^c$

As the value of each column in EHR is non-atomic (multiple data versions), the EHR tables satisfy NF^2. This property would suggest that it is possible to do a Cartesian product at any nested level. However, different than the general NF^2, the nested depth of any EHR table is fixed. This characteristic prohibits doing Cartesian product at any arbitrary nested level. In consequence, the output schema of $\boxtimes^c$ is as same as $\boxtimes^T$. We define the Cartesian product for EHR as:

$$r_1 \boxtimes^c r_2 = T_{TE}(T_{ET}(r_1) \boxtimes^T T_{ET}(r_2)),$$
$$\text{where the group key for } T_{TE} \text{ is composed by } r_1.rk \text{ and } r_2.rk. \tag{5.17}$$

The implementation of $r_1 \boxtimes^c r_2$ based on MapReduce is described in Algorithm 21. In the implementation, we assume the size of r_2 is less than r_1. In consequence, we broadcast r_2 in each Map Node (line 3). For each data version D_1 in r_1, we find all the data versions in r_2 which have TI overlapping with D_1 and are stored in s_2 (lines 4-9). Finally, we concatenate the row keys of D_1 and D_2 and emit the result data versions.

Note that we broadcast the smaller table among r_1 and r_2. However, it is possible that the small table is far larger than the capacity of a $Mapper$ node. In consequence, we can first indicate start and stop-row [1] to horizontally partition r_2. Then, we run the previous algorithm several times based on the number of partitions.

[1] For example, HBase provides *setStartRow*() and *setStopRow*() methods.

Algorithm 21 CTO Cartesian operation

```
1: Map(RowKey, RowResult)
2: Set s_2; // store data versions from r_2
3: Tuples tsets = Scan(r_2);
4: for each data versin D_1 in RowResult do
5:     for each data version D_2 in tsets do
6:         if overlap(D_1, D_2)≠ ∅ then
7:             s_2.add(D_2);
8:         end if
9:     end for
10:    if s_2 ≠ ∅ then
11:        emit(D_1.rk + D_2.rk, D_1);
12:        emit(D_1.rk + D_2.rk,s_2);
13:        s_2.setEmpty();
14:    end if
15: end for
```

5.3.7 Theta-join $\bowtie_p^c$

The definition of $\bowtie_p^c$ can be derived from $\boxtimes^c$ and σ_p^c.

5.3.8 Temporal aggregation $f_{A'}(\gamma^c_{(A,TG,W)})$

In the EHR context, tuples in an EHR table can be grouped based on:

1. Non-temporal attributes;
2. Temporal information;
3. Both.

The temporal aggregation based on EHR model are again classified as *Instantaneous temporal aggregation* and *Moving-window temporal aggregation*.

The definition of $f_{A'}(\gamma^c_{(A,TG,W)})$ is given as follows:

$$f_{A'}(\gamma^c_{(A,TG,W)})(r^c) = T_{TE}(f_{A'}(\gamma^T_{(A,TG,W)}(T_{ET}(r^c)))). \tag{5.18}$$

Notice that when non-temporal attributes (denoted as A) are not specified, there is no necessary to perform T_{TE}, as TI in the aggregation result is unique. Otherwise, the value of A will be used as row key for the output

Network

RK	Network:Supplier ttl=∞	Network:Speed ttl=∞
Tom	(1&1,[5,8)), (Telecom,[1,5))	(1270K,[5,8)), (920K,[4,5)), (1115K,[1,4))
Jim	(1&1,[2,8))	(1003K,[6,8)), (1100K,[2,6))

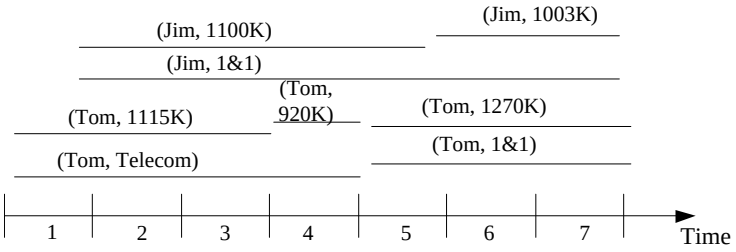


Figure 5.10: Tabular and graphical representation of Network

of T_{TE}. We utilize Query 4 and table "Network" in Figure 5.10 to illustrate the aforementioned description. For simplicity, we use notations "Supplier" and "Speeed" to denote columns "Network:Supplier" and "Network:Speed", respectively.

• **Query 3**: What is the daily average network speed of different suppliers?

The above query is specified as $AVG_{Speed}(\gamma^{c}_{(Supplier,DAY,0)})(Network)$ and the result is shown in Figure 5.11. The value of non-temporal grouped attribute "Supplier" (denoted as GP) is used as the row key of the aggregation result.

5.4 Output constraints of TTRO and CTO operators

To define the semantics of $\cup^T$, we made an assumption that no data conflict occurs, namely, it does not exist a tuple t_1 from table r_1 and a tuple t_2 from

GP	Res:AVG ttl=∞
Telecom	(920K,[4,5)), (1115K,[1,4))
1&1	(1136.5K,[6,8)), (1185K,[5,6)), (1100K,[2,5))

Figure 5.11: Results for CTO aggregation

r_2 where $(t_1.rk = t_2.rk) \wedge (t_1.value \neq t_2.value)$. However, when such conflict occurs, it is impossible to store t_1 and t_2 in a WCS based on the TTR model, as each WCS (TTR) table requires a unique row key. Although we can store the union results in a distributed file system, the union results do not satisfy the TTR model anymore. In consequence, when the temporal conflict exists, we cannot guarantee that the TTR tables are closed under the $\cup^T$ operator. The same thing can also happen to $\cup^c$ operator.

Based on the data model of WCS, the column-family names must be unique. Hence, it is impossible to store the results of self-join or self-Cartesian product in a WCS.

5.5 Performance evaluation

In this section, we compare the performance of the $TTRO$ and CTO variants to indicate which temporal operator model is more suitable for temporal data processing for WCSs. The experiments are performed on a six-node cluster (Xeon Quadcore CPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) running Hadoop (version 1.0.4) (Apache Hadoop) and HBase (version 0.94), respectively.

Test data sets are generated through the *workloada* of YCSB (Yahoo Cloud Service Benchmark) (YCSB) [YCS]. At the "Loading" phase, one million tuples are inserted into a predefined "usertable" in HBase. The "usertable" contains one column family named "cf1" with 10 columns and the "requestdistribution" property is set to uniform. After the loading phase,

the "usertable" has the implicit history representation and each column only contains a single data version. At the "Transaction" phase, we set the properties of "updateproportion" to 0.2 and "insertproportion" to 0. This configuration denotes that certain columns of 0.2 million tuples in the "usertable" will contain two data versions. The reason for this configuration is based on the consideration for studying the influence of different table representations against the temporal query performance. Then, we perform two table transformation operations, namely, T_{IE} and T_{IT} to generate the corresponding EHR and TTR tables. Table 5.1 summarizes the number of tuples after two table transformations.

Table name	**Table representation**	**Number of tuples**
usertable	Implicit-history representation	1 million
usertable_e	Explicit-history representation	1 million
usertable_t	Tuple-timestamping representation	1.2 million

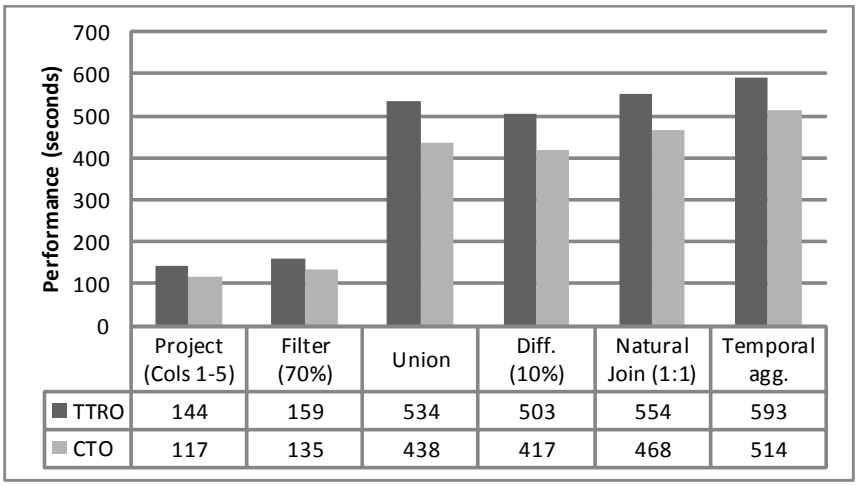
Table 5.1: Number of tuples from different table representations after various table transformations

From the above table, we can notice that "usertable_e" has the same number of tuples as "usertable". For table "usertable_t", the update data versions will cause the generation of 0.2 million tuples.

As we have already described in the previous chapter, when transforming the IHR table, the corresponding TTR table will have a simple tuple representation but usually contain more tuples (data redundancy) comparing the EHR table. Hence, it is worth to compare the performance based on these two table representations in a reasonable level. In an extreme situation, for example, when each tuple in "usertable" contains 100 data version, it implies that the number of tuples in "usertable_t" can be hundred times more than "usertable_e". In our performance comparisons, we do not consider this extreme situation.

As already described in the previous sections, each operator is implemented by utilizing MapReduce (Hadoop). *Unary* operators, such as Project (π_A^T and π_A^c) and Filter (σ_p^T and σ_p^c) can be implemented inside the Map function. *Binary* operators, such as Union ($\cup^T$ and $\cup^c$), Difference ($-^T$ and $-^c$), Intersection ($\cap^T$ and $\cap^c$), Join ($\bowtie_p^T$ and $\bowtie_p^c$) and temporal aggregations ($f_{A'}(\gamma_{(A,TI)}^T$ and $f_{A'}(\gamma_{(A,TI)}^c)$) need both Map and Reduce functions. We

implement Cartesian product ($\boxtimes^T$ and $\boxtimes^c$) with only the *Map* function by broadcasting the "small" table in each *Map* node.



	Project (Cols 1-5)	Filter (70%)	Union	Diff. (10%)	Natural Join (1:1)	Temporal agg.
TTRO	144	159	534	503	554	593
CTO	117	135	438	417	468	514

Figure 5.12: Performance comparisons between different temporal operator models

The performance of each *TTRO* operator and its corresponding *CTO* operator is depicted in Figure 5.12. The picture shows the performance of each individual temporal comparator over the test data. For every temporal operator, *CTO* outperforms *TTRO* even though *CTO* needs more complicated implementation logic.

For the project processing, *TTRO* will read 1.2 million tuples, output the columns from 1 to 5 and insert 1.2 million tuples into the result table. Comparing to *TTRO*, the *CTO* project operator only needs to read and insert 1 million tuples. For the filter processing, 70% of the tuples satisfy the selection predicate. Hence, *TTRO* will process 2.04 million tuples (read 1.2 million tuples and insert 0.84 million tuples) where *CTO* will only process 1.7 million tuples (read 1 million tuples and insert 0.7 million tuples).

For the binary operation, such as *Union*, *Difference* and *Natural join*, as the Reduce phase is needed, the *TTRO* data processing needs more data-shipping time comparing to the *CTO*. For example, the *TTRO* Union operator will first scan 2.4 million tuples from two tables (in our performance test, we implement "usertable_t" Union "usertable_t"). After the *Map* function finishes, these 2.4 million tuples will be delivered to the corresponding

Reduce node.

In the performance chart, "10%" for the *Difference* operator denotes that 10% of the tuples from "usertable_t" or "usertable_e" will be discarded and "1:1" indicates that two join tables have a one to one relationship. For the temporal aggregation, the *Instantaneous temporal aggregation* with $AVG()$ is evaluated.

In HBase, tuples are represented as key-value pairs stored in so-called "HFiles". Each HFile contains several data blocks and the length of each data block is fixed (in default 64K). In consequence, to store the *same* information, TTR will spend more I/O costs than EHR because of data redundancy. The term "same" means the TTR table and the EHR table are derived from the same IHR table.

To handle the *Put* commands (result-tuple insertions) in HBase, when the result table is empty, it will first route all the Put commands to the same node. When that node reaches the configuration threshold or when all the Put operations are done, HBase will automatically re-balance the data in the cluster. During the data re-balancing, the corresponding data nodes will then be off-line until the re-balance task is finished. Moreover, as HBase is memory-intensive, when Garbage Collection is executed by the JVM (Java Virtual Machine) during loading the results, any Put requests are stalled and the server will pause for a significant amount of time. As the IHR table contains more data volume than the EHR table, processing an IHR table has a higher probability to trigger data re-balance processing and JVM Garbage collection.

5.6 Summary

As we have already described in Chapter 4, IHR (implicit-history representation) can cause wrong or misleading results during the temporal query processing because of its implicit TI strategy. In consequence, to store the temporal data in DW based on WCSs, we can either adopt EHR (explicit-history representation) or TTR (tuple-timestamping representation).

For processing TTR tables, we introduced the $TTRO$ operator model as a minor extension of the temporal relational algebra. For processing EHR tables, we proposed a novel temporal operator model called CTO. The CTO model can be directly applied to the EHR tables without requiring the additional table restructuring operations. Both $TTRO$ and CTO include eight

temporal operators, such as *Union*, *Difference*, *Intersection*, *Project*, *Filter*, *Cartesian product*, *Theta-Join* and *Group-by* with a set of aggregation functions, such as SUM, AVG, MAX and etc.

The TTR and EHR tables are closed under the $TTRO$ model and the CTO model, respectively, when no temporal conflicts exist. Otherwise, the TTR and EHR are not closed under $\cup^T$ and $\cup^c$.

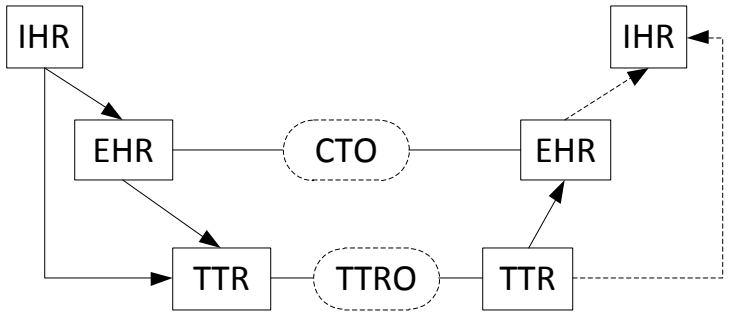


Figure 5.13: Data processing and table transformation stack

Figure 5.13 shows an overview of the temporal data processing and table transformation stack which is a refinement of Figure 4.7. To process the temporal data in WCSs, users can either write a script using CTO or $TTRO$ operators (the same strategy used by Pig Latin) or a SQL-like language can be built on top of CTO or $TTRO$.

The performance chart shows that the CTO operator model outperforms the $TTRO$ operator model for the temporal query processing. The reason for that is that the EHR contains less redundant data than TTR and the time for table-scanning and result-tuple-insertion dominate the temporal query processing when the temporal data volume is large.

Chapter 6

Incremental data recomputation - state of art

In this chapter, we first describe the motivation of incremental data recomputation and discuss how it can influence query response time. Then, we discuss how incremental recomputation can be used in different contexts, namely, for non-temporal databases and for temporal databases.

6.1 Motivation

When analyzing data, people usually specify a set of functions or rules to obtain the desired results. For example, users utilize the SQL language to process data stored in relational database systems. After query processing is done, the query results are usually stored "*temporarily*" (In the relational database context, the query results can be treated as *views* [UW97]). The term "temporarily" here means:

- When query results are generated, they will only exist in the database system for a short period of time.
- Query results are not persistent on disk and will get lost when, e.g. database systems restart after database failures occur.
- No log entries in the database system will record query results.

Often, the same data analysis tasks are periodically repeated, while the state of base data changes over time. In consequence, it may be better to

cache or materialize query results to reduce query response time. To incorporate changes made at the base data, a complete query re-evaluation can be performed (also known as "start-from-scratch"). However, this approach is time-consuming and inefficient when the size of change data (deltas) is much smaller than the size of the whole base data. The more cost-effective way is *incremental recomputation* [GM95, LMSS95, Gea95, BkLLT86, QW91, ZGMHW95, JD09] by only propagating the deltas into the persistent query results without re-evaluating unchanged data.

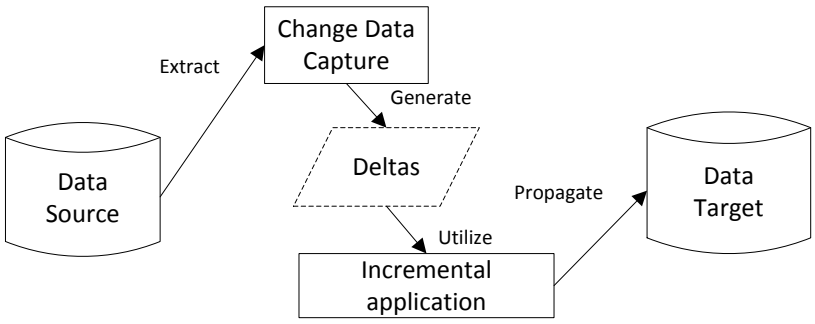


Figure 6.1: Data flow of the incremental recomputation

Figure 6.1 describes the relationships between data source, change-data capture (CDC), incremental application and data target. At first, CDC detects and extracts data changes made at the data source. Then, the incremental application utilizes the deltas to refresh the state of the data target.

In the relational database context, incremental recomputation has been extensively studied and widely used to maintain so called "*Materialized views*" [GM95, LMSS95, Gea95, BkLLT86, QW91]. In the ETL/data warehouse [KC04] context, incremental recomputation is also adopted to evaluate ETL processing and refresh the state of data warehouses [ZGMHW95, LGm96b, JD08, JD09].

In the following sections, we will first explain how incremental recomputation can be utilized in the non-temporal data context. Then, we describe the approaches which can be exploited to incrementally re-evaluate temporal query results [YW98, YW00, JMRS93, Bea95].

6.2 Incremental recomputation for non-temporal data

In this section, we mainly focus on how incremental recomputation can be applied to non-temporal data. We first describe approaches for materialized view maintenance. Then, we illustrate how incremental recomputation can be used in the ETL/Data warehousing (DW) context. Finally, we introduce the approaches and frameworks which are currently used by large companies for accelerating "Big data" processing.

6.2.1 Materialized view maintenance

In a relational database system, a *view* is a virtual table which is derived from a set of base tables. In general, users provide view definitions based on relational algebra. *Materialized views* are a specific type of views in which view data is persistent on disk. Although people can access materialized views as normal tables, their content becomes stale when the underlying base relations are changed. In consequence, materialized views need to be updated to reflect the changes made at the base data.

To maintain the results of materialized views, two approaches can be adopted. The first one is full recomputation which reevaluates the complete query provided in the view definition. The second one is incremental recomputation which only propagates the data changes from base relations to the materialized views without accessing and querying the unchanged data sets. For performance reason, the latter approach is preferable, when the size of the base relation is much larger than the size of the change data.

Incremental view maintenance has been extensively studied for decades [GM95, LMSS95, Gea95, BkLLT86, QW91]. Generally, two methods can be followed:

- **Algorithmic approach**: Designing specific algorithms for view maintenance.
  Example 1: A *Counting algorithm* [BkLLT86] applies to SQL views by explicitly or implicitly associating each derived tuple in the materialized view with its frequency of occurrence. A typical use case of the counting algorithm is to incrementally maintain a view which includes the project operator. As the project operator (in a set context when "DISTINCT" is specified) will eliminate data duplicates, it is

impossible to deduce the correct state of the view without additional information. For data changes, insertions are represented with positive counts and deletions with negative counts. To update the materialized view, positive counts are added in and negative counts are subtracted. A tuple with zero counts is discarded.

- **Algebraic approach** (also called *algebraic differencing approach*): Treating the view definition as a mathematical formula and deducing incremental change data propagation equations (rules) [QW91, Gea95, GM95]. Deltas are classified as insertions Δ and deletions ∇ where an update is modeled as a delete-insert pair. The new state of a view is computed by discarding the deleted tuples and inserting the newly generated ones, i.e. $V^{new} = V^{old} - \nabla \cup \Delta$.
  Example 2: Consider a view $Stu - dep$ that is defined as an $equi - join$ of relations *Student (stu_no, stu_name, dep_no)* and *Department (dep_no, building)*:

$$Stu - dep = Student \bowtie_{dep\_no} Department \tag{6.1}$$

  Let $\Delta(Student)$ and $\Delta(Department)$ represent the tuples which are inserted into relations *Student* and *Department*, respectively. The change data propagation rule for $equi - join$ based on insertions is

$$(R \cup \Delta R) \bowtie_p S = (R \bowtie_p S) \cup (\Delta R \bowtie_p S) \tag{6.2}$$

  The corresponding modifications for view $Stu-dep$ can be computed by recursively applying the change data propagation rule until the original view definition is met:

$$\begin{aligned}&(Student \cup \Delta Student) \bowtie_{dep\_no} (Department \cup \Delta Department) = \\ &\underbrace{(Student \bowtie_{dep\_no} Department)}_{\text{Original view definition}} \cup (\Delta Student \bowtie_{dep\_no} Department) \cup \\ &(Student \bowtie_{dep\_no} \Delta Department) \cup (\Delta Student \bowtie_{dep\_no} \Delta Department)\end{aligned} \tag{6.3}$$

Comparing these two approaches, the first one (algorithmic approach) is used to deal with a specific incremental maintenance problem and in those cases is more effective than the second approach. For example, in general, the algebraic approach cannot deduce the incremental change data propagation rules for the project operator based on the set context. However, the soundness of the algorithm approach is hard to prove and it is difficult for the database system to optimize.

6.2.2 Remote materialized view maintenance

A data warehouse (DW) is a data repository which stores integrated data from one or more than one data source. Abstractly, the tables in a DW can be considered as materialized views which are derived from multiple source tables. However, different from the discussions made in the previous section, a DW is usually an autonomous system which resides on different machines than its data sources.

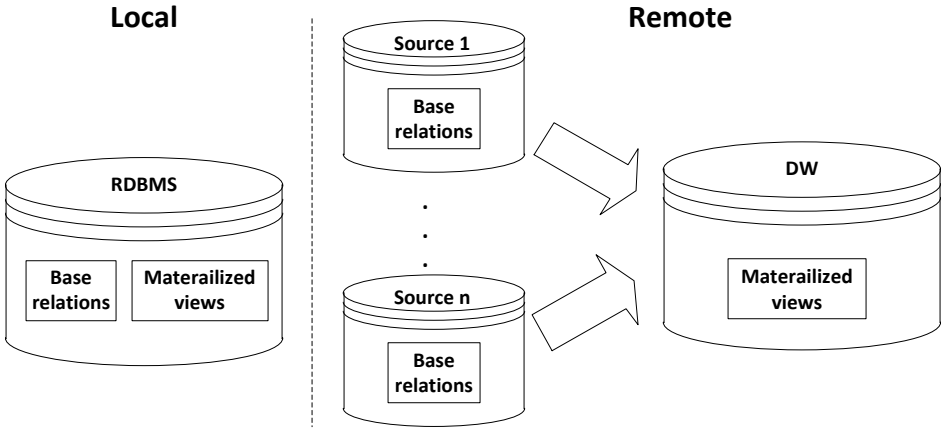


Figure 6.2: View maintenance in a local and a remote manner

Figure 6.2 describes the differences between view maintenance in a local and a remote manner. The left side illustrates the traditional view maintenance. It assumes that both base relations and materialized views are under the control of a single database system. In the remote manner, base relations and materialized views are managed by different systems residing on different machines connected via a network.

When incrementally maintaining the tables in a DW, the traditional view maintenance approaches have to deal with so called *distributed incremental view maintenance anomalies* [ZGMHW95, JD09] or *maintenance anomalies* in short. Maintenance anomalies occur as DW and data sources are both autonomous systems. They happen when DW tries to update its state while data sources are changing. Although data sources can report their changes periodically or directly to the DW, as they generally do not understand the view definition, the DW has to issue queries back to certain data sources.

However, it is possible that data sources get modified while queries are sent back to the sources. In consequence, maintenance anomalies occur.

To overcome maintenance anomalies, several approaches are proposed:

1. *Distributed transaction.* Putting change data reporting, queries send back and view reevaluation in a single distributed transaction. Because of the isolation property, no inconsistent states of source tables will be read.

2. *Source table duplication.* The main reason for the maintenance anomalies is that a DW has to send queries back to source tables to accomplish view maintenance. To avoid that, the DW can make copies of source tables based on the view definition. However, such an approach can cause a considerable overhead. Moreover, the copies of source tables in the DW or staging area also need to be updated in response to the data modifications made at the data sources.

3. *Compensating algorithm.* Instead of explicitly issuing distributed transaction or making copies of relations involved in view definitions, authors in [ZGMHW95] utilize so-called *compensating queries* to avoid maintenance anomalies. The basic idea is to send compensating queries together with the usual queries back to data sources to offset the effects of concurrent updates. However, compensating queries can only produce correct results when we assume the messages between data sources and the DW are delivered in order and data sources have the ability to evaluate the queries.

6.2.3 Incremental ETL processing

ETL [KC04] is a shorthand for *Extract*, *Transform* and *Load*. In general, it describes a series of data processing steps, namely, extracting data from homogeneous or heterogeneous data sources, transforming extracted data by applying a set of rules or functions to generate proper format or structure and loading the transformed results into the final target e.g. a file system or data warehouse. Generally speaking, ETL processing can be considered as a style of materialized data integration.

To incrementally recompute ETL processing, the change data propagation rules which are proposed for materialized view maintenance can be

reused when ETL graphs (or processing operators) are represented by relational algebra, namely, *Algebraic differencing approaches* [JD08, JD09]. However, as ETL tools usually access data from heterogeneous data sources, data cleansing is a very important task. Due to a study made by [Jör13], incremental ETL jobs may not even outperform the naive full recomputation as it usually involves data cleansing operations. To accelerate incremental ETL jobs, [Jör13] leverages the notion of safe delta propagation. Safe deltas are a superset of true deltas and may be computed more efficiently with data cleansing operators.

Generally, change data in view maintenance is classified as insertion and deletion where update is represented as a deletion/insertion pair. In the ETL/DW context, a so called "Change-Data Capture" (CDC) is used to detect data changes made at the data sources. In practice, different CDC approaches can be utilized and not all of them can produce complete deltas. Hence, [JD11] proposes approaches for incremental ETL processing with partial change data (deltas) for a specific type of view, namely *Dimension Views*. Dimension views are SPJ (Select-Project-Join) views in which each tuple in the view must contain a unique identifier and the join predicates at least references the primary key of one base table.

The partial delta sets are classified as *insertion*, *deletion*, *updates*, *partial updates*, *upserts* and *partial deletions*. For propagating partial deltas, the change data propagation rules proposed by [Gea95] are modified w.r.t. the type of partial deltas. Moreover, a data target is assumed to have the ability to handle "partial" modifications, e.g. to support a *MERGE* operation when "upserted" deltas are propagated.

6.2.4 Incremental recomputation for "Big data" processing

Incremental recomputation attracts more and more attention in the "Big data" context, as it can heavily reduce query response time. In the following, we introduce several approaches or frameworks which attempt to implement incremental recomputation.

- *Incoop* [BWR+11] was developed by Max Planck Institute. It extends the Apache Hadoop framework in several ways. First, HDFS (Hadoop distributed file system) is extended to *Inc-HDFS* which provides the ability to identify similarities in the input data. Secondly, large task

will be divided into a set of smaller tasks with a controlled granularity. Thirdly, a new MapReduce scheduler called *Memorization-aware scheduler* is proposed. The Memorization-aware scheduler will assign the data analyzing tasks only to the needed machines (the machines which contain the change data).

- *DryadInc* [PBYI09] is a system which is developed by UC Berkley and Microsoft Research. To incrementally maintain query results, DryadInc caches intermediate results from prior query executions and reuses them when they occur unchanged in future computations.

- *Percolator* [PD10] is a system which is built by Google for processing large data sets incrementally. Different from MapReduce, Percolator is built on top of Google's Big table. It provides a new programming model called "Observer". An observer is similar to a trigger in relational database systems. It consists of a piece of code which is invoked when a specific event occurs.

- *CBP* (Continuous bulk processing) [LOR+10] is a system which is developed by UC San Diego and Yahoo Research. CBP is inspired by MapReduce and introduces several new primitives which can be used to store and reuse prior results for incremental recomputation. In each data transformation step, the input records are divided into several disjoint partitions and certain criteria can be specified to determine the runnability of a transform step depending on its available input data.

- *Marimba* [JPYD11, SJHD14] is a framework which is proposed by TU Kaiserslautern and addresses the maintainability of a specific MapReduce jobs, namely, *aggregate self-maintainable MapReduce jobs* for a wide range of data transformation and analysis tasks. The term "self - maintainable" denotes a specific type of materialized views (or query results) in which the state of materialized views can be maintained by only using the data changes and view itself. Marimba extends MapReduce by adding new MapReduce classes to provide the systematic support for incremental MapReduce jobs. To deduce incremental MapReduce jobs, each aggregate self-maintainable MapReduce job is first abstractly represented as an *Abelian group*. The new query results are calculated based on the types of the aggregation functions and the neutral element of that Abelian group.

To summarize the previous approaches, three categories can be classified.

1. *Incoop* and *DryadInc* use caching mechanisms. They attempt to reuse the previous results. Although this approach is transparent to users and no new programming model will be introduced, the overhead of storage will be extremely high.

2. *Percolator* and *CBP* introduce new programming model primitives for incremental recomputations. Although the storage overhead is avoided, it requires programming skills and forces users to work at a very low level of abstraction.

3. *Marimba* is inspired by the approaches of materialized views. It maps each aggregate self-maintainable MapReduce jobs into an Abelian group. Comparing to the former two categories, Marimba does not require to cache the intermediate results or force programmers to use new programming paradigms.

6.3 Incremental recomputation for temporal data

Although incremental recomputation has been extensively studied for non-temporal data for a long time, incremental temporal data processing has only been addressed on some aspects. In general, incremental recomputation based on temporal data can be tackled in two ways:

1. Transforming temporal data (including base data and change data) and temporal queries into their non-temporal counterparts and reusing the approaches which are designed for non-temporal data [JMRS93].

2. Propagating temporal data changes directly based on the predefined temporal operators [YW98, YW00, Bea95].

6.3.1 Reusing non-temporal approaches for incrementally recomputing temporal data

To guarantee the correctness of a temporal query when utilizing non-temporal operators, e.g. relational algebra, people have proposed a notion called

"*Snapshot Reducibility*". Snapshot Reducibility describes that a temporal query applied to the temporal relations can be evaluated based on each individual snapshot of those relations at each distinct time instant.

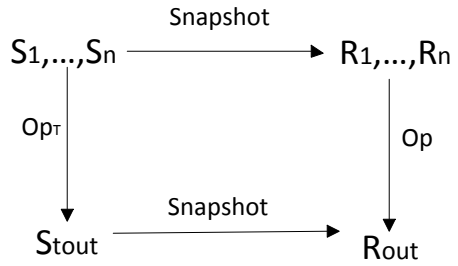


Figure 6.3: Snapshot Reducibility

Figure 6.3 gives a graphical illustration of the Snapshot Reducibility. $S_1, ..., S_n$, Op_T and S_{tout} on the left denote temporal tables, temporal queries and temporal results, respectively. $R_1, ...R_n$, Op and R_{out} on the right indicate the corresponding non-temporal data, non-temporal queries and non-temporal results, respectively. To connect these two areas, Snapshots of each S_i (where $1 \leq i \leq n$) will be taken and temporal operators (queries) Op_T will be transformed to Op.

Authors in [JMRS93] propose an incremental implementation model for relational databases with transaction time based on the notion of "Snapshot Reducibility". Each table R in such model is correlated with a so called "*Backlog*" B_R. A Backlog B_R stores the complete history of data modifications of table R. In general, the schema of a Backlog B_R consists of all the attributes of its corresponding relation R with additional three columns, namely, Id, Op and $Time$.

Id can be considered as a surrogate key holding an automatically generated value. Op denotes what kind of data modification, namely, whether an insertion, a deletion or a modification (update) was performed. $Time$ indicates the time instant when data modification is done, i.e. the time when its corresponding transaction was committed.

With the help of B_R, the state of a relation R at each distinct time instant (each snapshot of R) can be represented as a vector (or an array) in which each element in that vector is a pointer to a tuple of its Backlog. In consequence, temporal results can be rebuilt based on its corresponding

non-temporal results by using the Backlog.

To incrementally calculate temporal results based on non-temporal operators and Backlogs, the following steps will be performed:

1. Transforming temporal data (including both base relations and deltas) and temporal queries into their non-temporal counterparts.

2. Reusing non-temporal incremental approaches, e.g. the strategies for materialized view maintenance.

3. Utilizing Backlogs to transform non-temporal results to temporal results.

Although this approach can be seamlessly integrated with the existing relational database systems, storing the whole data-change history (Backlog) for each table will cause a huge storage overhead.

6.3.2 Propagating temporal deltas with temporal operators

For a long time, there was no official standard for the temporal databases [1]. Different research groups and companies proposed their own temporal data models and temporal operators. Hence, in the following, we will introduce approaches for incremental temporal recomputation based on various temporal data models and temporal operators.

View maintenance for the chronicle data model

A *transaction recording system* is a system which records a stream of transaction information, such as telephone calls, stock trades and so on. To meet the stringent performance requirement of such a system, AT&T Bell Laboratories [JMS95] proposed a new temporal data model called *Chronicle data model* to store transactional data.

A chronicle is similar to a relation, except that each tuple in a chronicle is associated with a unique sequence. Normally, a sequence number is generated by a chronicle database systems to guarantee that the sequence number of the

[1] Although SQL 2011 introduces the temporal features, not all companies implement them.

new generated tuples are greater than all existing ones. Moreover, each tuple in a chronicle is associated with an explicit temporal instant (or chronon) to indicate when that tuple was created.

Different from a normal relation, a chronicle can only be modified by appending new tuples at the end of it. Data modifications such as update and deletion are not supported.

To query a chronicle, a new temporal algebra called "*Summarized Chronicle Algebra*" is defined. The summarized chronicle algebra can be considered as an extension of relational algebra. It contains seven operators, such as *Select*, *Project*, *Natural equijoin*, *Union*, *Difference*, *Group-by with aggregations* and *Cartesian product*. The semantics of each operator is constrained by the sequencing information, e.g. a Cartesian product of two chronicles R and S implicitly requires a temporal join on the sequencing numbers.

Authors in [JMS95] exploit the algebraic differencing method to deduce temporal delta propagation rules based on the summarized chronicle algebra. Due to the data modifications supported by the chronicle model, the updates of chronicles can be represented as $C^{new} = C^{old} \cup \Delta C$. Moreover, as the chronicle model guarantees the sequence number of all inserted tuples will be larger than existing ones, there will be no sequence-number equalities between the old state of one chronicle and the new state of an other one. Consequently, the change data propagation rules for binary operators are simplified. For example, $C_1 \bowtie_p C_2$ will only be updated when $\Delta C_1 \bowtie \Delta C_2 \neq \emptyset$ (as both $\Delta C_1 \bowtie C_2$ and $\Delta C_2 \bowtie C_1$ are empty).

Temporal view maintenance with temporal relational model

A *temporal data warehouse* is a data repository which stores temporal and time-related data for data analysis. A temporal data warehouse can be utilized to e.g. keep track of marketing data, fraud detecting and personal information. Each table in a temporal data warehouse can be considered as a materialized temporal view which is derived from various data sources. Note that data sources can be non-temporal database systems.

To model a temporal table in a data warehouse, researchers [YW98] at Stanford University proposed a *Bitemporal Conceptual Data Model* (BCDM). The BCDM model can be seen as an extension of the relational data model in which each tuple is associated with an explicit temporal interval (TI). However, different from the temporal data model used in *SQL 2011 standard*, tuples in the same table which are value-equivalent will be represented as a

single tuple in which the temporal interval (TI) of such tuple is a set of TIs which denote when that tuple is valid. Moreover, the BCDM model requires the TI to be included in the primary key.

The temporal query operators (for the BCDM model) are extensions of the relational algebra. Six operators, such as *Difference*, *Union*, *Projection*, *Selection*, *Join* and *Cumulative temporal aggregation* are defined. For Union and Difference, duplicates or equivalent tuples will be removed when they share the same values and are valid during the same period of time. For the join operator, two tuples will be joined together when they satisfy the join predicate and are valid during the same period of time.

To deduce the temporal change data propagation rules, temporal deltas are represented as inserted and deleted sets and temporal queries are classified into different categories based on their semantics. Before propagating temporal deltas, temporal queries are classified into τ-Reducible and θ-Reducible. τ-Reducible denotes queries that have no explicit reference to temporal attributes and θ-Reducible indicates that temporal attributes are referenced in the query definition. Based on different types of temporal queries, the corresponding temporal deltas need to be generated. The rules for propagating temporal deltas looks the same as the rules for their non-temporal counterparts [QW91].

Although this approach can propagate data changes incrementally, transforming the suitable delta content based on the semantics of temporal queries is cumbersome and time-consuming.

Temporal aggregation

In the non-temporal database context, view maintenance with aggregation functions has been studied for a long time, see survey [GM95].

In the temporal database context, several approaches [KS95, YW03] addressed the incremental recomputations based on *instantaneous temporal aggregation* (ITA) and *moving-window or cumulative temporal aggregation* (CTA). Generally, temporal deltas are classified as deletion ∇ and insertion Δ where an update is represented as a deletion/insertion pair. To implement the incremental recomputation, new index structures [KS95, YW03] are introduced based on memory and disk.

[KS95] introduces an index structure called *aggregation tree* to achieve the incremental recomputation based on ITA. The aggregation tree is an alternative of segment tree [THC09]. It stores in main memory and maintains

the hierarchy of TIs and partial aggregation results. To maintain the aggregation tree, the starting- and ending-points of TIs in each inserted tuple are used to split the tree. However, the paper lacks the discussion about how to main the aggregation tree with deletions.

[YW03] proposes a disk-based index called *SB-tree* to incrementally calculate CTA. A SB-tree combines the features of segment tree and B-tree [THC09]. It stores a hierarchy of TIs and partial aggregation results. As same as B-tree, each SB-tree is associated with an order b. b denotes that each SB-tree node (except root node) should at least contains $b - 1$ keys (time points) but not more than $2b$. When a tuple is inserted, the TI of that tuple is used to split the tree. The processing of SB-tree splitting is as same as B-tree. When deletions occur, they will be treated as insertions with "negative" effect on the aggregation values. After SB-tree is updated, a *compact* processing will be performed to merge intervals of the tree nodes.

For both approaches, as the index is built for the aggregation values, the results of aggregation functions MIN and MAX cannot be incrementally computed.

Chapter 7

Temporal change-data capture

Change-data capture (CDC) [KC04] is a term used mostly in the data replication and ETL/DW context. It is a prerequisite task for incremental recomputation and describes process of extracting data changes (deltas) made at data sources. In this chapter, we illustrate how temporal deltas can be modeled and extracted from data sources, i.e. wide-column stores (WCSs).

In general, CDC processing will include the following steps:

1. Detecting deltas.
2. Classifying delta items into appropriate categories based on the modification events.
3. Customizing delta outputs based on the application logic and the data targets.
4. Delivering deltas.

As we will see in subsequent sections, each of these steps is influenced and constrained by *peculiarities of the data sources* and *characteristics of the incremental applications*. For example, to incrementally calculate the result of an aggregation function, such as SUM, CDC should produce the before and after state of updated tuples.

As we have already seen in Chapter 2, in contrast to relational database systems, WCSs do not distinguish between update and insertion. New data versions (values) will be generated through "*Put*" commands. When issuing a "Put" command, users need to denote the parameters such as row key,

column-family name, column name, value and timestamp (optional). Following the data model of WCS, data deletions can partially prune a row by eliminating column values at various granularities, namely, *data version*, *column* and *column family*. A row deletion will be represented as a number of column-family deletions. Different from relational database systems, the delete operations in WCSs will not delete data right away but insert a marker called "tombstone" to mask the data values whose TSs are less than or equal to the TS of the tombstone. The actual data deletions take place at data compaction time.

Although WCSs allow to manually assign and modify the value of TS, it is not recommended by the WCS community [Hba]. As the timestamp (TS) may be internally used by WCSs for some tasks, e.g. ttl (time-to-live) calculation and stale data version deletion, manually modifying a TS without careful considerations can easily cause confusion and unexpected behavior, e.g. inserting an invisible tuple (if the new inserted tuple does not pass the ttl constraint). In consequence, we assume that the TSs have the semantics of *transaction time* and the values of TSs are automatically generated by the database systems (WCSs).

Recall the architecture of a temporal data warehouse (DW) in Figure 5.1. Each data value in the data sources is associated with a TS where each tuple (or data version) in the DW is associated with a temporal interval (TI). Hence, it is the responsibility of CDC to translate the TS to a TI for each delta item (we will illustrate how such translation can be done in Section 7.1).

In the following sections, we will first depict our logical temporal delta model and its corresponding physical representations. Then, we describe five feasible CDC approaches, namely, *Timestamp-based approach*, *Audit-column approach*, *Log-based approach*, *Trigger-based approach* and *Snapshot differential approach* and indicate their advantages and drawbacks. Finally, we compare their performance.

7.1 Temporal delta model

In general, the design and the implementation of CDC is influenced by the peculiarities of data sources and the characteristics of incremental applications. Maintenance of materialized views in relational databases (RDBMSs) usually require a complete delta set and both data sources and data tar-

gets are relational tables. Due to the data manipulation language (DML) supported by RDBMS, change data is usually classified into insert deltas and delete deltas (update deltas are represented as pairs of delete deltas and insert deltas).

To incrementally load and recompute the transformation results in the ETL/DW context, the strategies for maintaining materialized views are then utilized in distributed environments. Normally, the captured deltas are often "*partial deltas*" [JD11, HD13] due to

1. the data processing constraints of CDC approaches,
2. the data access restrictions of data sources, and
3. the efficiency of change data extraction mechanisms.

Partial deltas may lack attribute values, e.g. the previous state of an updated tuple is unknown, or the type of partial deltas may be uncertain, e.g. inserted tuples cannot be distinguished from updated ones. Although partial delta lacks some kinds of information and cannot be utilized by maintaining materialized view techniques, it is very useful to maintain an important type of tables, i.e. dimension tables in DW [KC04].

Before going into the details of the logical delta model, let us indicate what kinds of delta information is provided by WCSs right way. For recovery concerns, each data node in a WCS maintains log information to record the data modifications. As we have already seen in Chapter 2, WCSs support two sorts of data modifications, namely, a "Put" command which generates a new data version for a denoted column and a "Delete" command which produces a "tombstone" to mask the deleted data values. When any of these operations occurs, it will be recorded in a log entry. Clearly, deltas provided by WCSs are partial for following reasons.

1. The data modification type of a delta is ambiguous, i.e. an update cannot be distinguished from insertion.
2. Before images of deleted data values are missing, i.e. a tombstone is only a tag for masking the deleted data values.

Different from the general ETL/DW scenario, the partial nature of deltas is caused by the characteristics of WCSs (e.g. DML operation semantics) rather than a specific CDC approach or the access restrictions of data sources.

Obviously, the usage of such partial deltas sets is very limited. For example, as an insertion is not differentiated from an update, it is impossible to incrementally calculate the aggregate values. Moreover, as each tuple (or data version) in the DW contains an explicit temporal interval (TI), it is unclear how each delta item extracted from the data source will affect the TIs in the DW.

Hence, the aforementioned partial delta information has to be further refined and a suitable delta model needs to be built. In the following sections, we propose our logical delta model and describe the corresponding physical delta representations.

7.1.1 Logical delta model

To generate delta sets which can be used by different incremental applications, each delta object derived from WCSs is modeled as a quadruplet: (*identifier*, *operation*, *time-dimension*, *value*).

- **Identifier** includes information such as row key, column-family name and column name which are derived from the source data.
- **Operation** indicates the type of data modification which was performed. As WCSs do not distinguish between insertion and update and supports various deletion granularities, the original (basic) operation types supported by WCSs are "Put", "Delete" (version deletion), "DeleteColumn" and "DeleteFamily". However, these operation types reflect an incomplete delta set and lack the ability to work with diverse incremental applications, e.g. a procedure which incrementally maintains aggregations cannot use the delta sets with "Put" tag. Hence, we further refine and extend the basic operation types into $Insert$, $Update_{old/new}$, $Update_{partial}$, $Upsert$ and $Delete_{old/new}$. $Insert$ indicates that deltas are generated by insertion; $Update_{old/new}$ denotes deltas generated by updates and represents the state before and after an update, respectively; $Update_{partial}$ only indicates the state after an update; $Upsert$ denotes the delta is produced via insertion or update. Different from traditional database systems, WCSs support data deletions in multiple granularities. Hence, it is possible that the tuple (columns) still contain some data values after data deletions. $Delete_{old/new}$ denotes that deltas are generated through deletions and depict the deleted data values and the remaining data values after deletion, respectively.

- **Time-dimension** contains an operation timestamp (OP_{TS}) which denotes when a specific data modification occurs, a data version (tuple) timestamp (Val_{TS}) which describes when that data version is generated and a temporal interval (TI) which indicates how long a data version (tuple) is valid.

- **Value** represents the delta value based on the sort of data modification.

It is apparent that our logical delta model represents deltas at various completeness levels. We refer to deltas which are incomplete as partial deltas. A delta object may be partial delta because

1. the operation type is ambiguous, e.g. the delta operation type is Upsert, or

2. the delta value is incomplete, e.g. a delta object which only contains the after state for an update operation.

We give a formal definition for partial deltas for WCSs as follows:

Definition 7.1.1 (Partial Deltas). Let $R(rk, cfs : cols)$ be a WCS table with row key *rk* and a set of *cfs:cols* (column families:columns). For one *rk*, each column may contain multiple data versions. Let R_{old} denote the state of R at time point t_1 and R_{new} the state of R at time point t_2, where $t_1 < t_2$. During t_1 and t_2, several data modifications take place. Partial deltas of R during time span t_1 and t_2 are a five-tuple of sets (R_{ins}, $R_{upn/upo}$, R_{pup}, R_{ups}, $R_{delo/deln}$) where:

- $R_{ins} \subseteq R_{new}$ indicates a set of data values inserted into R; (**insertion**)

- $R_{upo/upn}$ with $R_{upn} \subseteq R_{new}$ and $R_{upo} \subseteq R_{old}$ denotes a set of tuples updated in R; (**update**)

- $R_{pup} \subseteq R_{new}$ denotes a set of data values updated in R only with the current state; (**partial update**)

- $R_{ups} \subseteq R_{new}$ indicates a set of data values either inserted or updated in R; (**upsert**)

- $R_{delo/deln}$ with $R_{delo} \subseteq R_{old}$ and $R_{deln} \subseteq R_{new}$ indicates a set of data values deleted from R and the rest of data values remaining in R, respectively; (**deletion**)

Based on the description of our temporal data warehouse (DW) architecture in Section 5.1, each data value in the data source is associated with a TS whereas each tuple (or data version) in DW carries in an explicit TI. CDC therefore has to explicitly translate the TS of each delta item into its corresponding TI. Moreover, each base relation in the data warehouse maintains a version history and no current data will be ever deleted or overwritten. The delta transformations are described as follows.

Definition 7.1.2 (Delta transformations). Let R^I and R^T denote a source table and its corresponding base relation in DW. We utilize R^I_{ins}, $R^I_{upo/upn}$, R^I_{ups}, R^I_{pup} $R^I_{delo/deln}$, R^T_{ins}, $R^T_{upo/upn}$, R^T_{ups} and R^T_{pup} to denote insert, update, upsert, partial update and delete delta sets of R^I and R^T, respectively. The deltas transformation between these two delta sets are given as follows:

- $R^I_{ins} \rightarrow R^T_{ins}$: For every tuple (version) s in R^I_{ins}, TI of the corresponding tuple s' in R^T_{ins} is represented as $[s.TS, End)$ where End is ∞ or calculated by ttl.

- $R^I_{upo/upn} \rightarrow R^T_{upo/upn} \& R^T_{ins}$: An update occurs at R^I will cause a tuple insertion and a TI update of the corresponding tuples in R^T. For each tuple s in R^I_{upn}, TI of the corresponding s' in R^T_{ins} is represented as $[s.TS, End)$. For each tuple t in R^I_{upo}, it is translated to a TI update (t_1, t_2) in $R^T_{upo/upn}$. $t_1.TI = [t.TS, End)$ and $t_2.TI = [t.TS, Up_{TS})$ where Op_{TS} denotes that when the update occurred.

- $R^I_{ups} \rightarrow R^T_{ups}$: For every tuple (version) s in R^I_{ups}, TI of the corresponding tuple s' in R^T_{ups} is represented as $[s.TS, End)$.

- $R^I_{pup} \rightarrow R^T_{pup}$: For every tuple (version) s in R^I_{pup}, TI of the corresponding tuple s' in R^T_{pup} is represented as $[s.TS, End)$.

- $R^I_{delo/deln} \rightarrow R^T_{upo/upn} \& R^T_{ins}$: As base relations in DW store the complete historical data, no current data will be deleted. In consequence, a deletion occurred at R^I will be translated as a TI modification and a tuple insertion. For each tuple s in R^I_{deln}, TI of the corresponding s' in R^T_{ins} is represented as $[Del_{TS}, End)$, where Del_{TS} denotes that when the deletion occurred. For each tuple t in R^I_{delo}, it is translated to a TI update (t_1, t_2) in $R^T_{upo/upn}$. $t_1.TI = [t.TS, End)$ and $t_2.TI = [t.TS, Del_{TS})$.

Based on the delta transformations described above, partial deltas defined in Definition 7.1.2 can be refined as a four-tuple of sets (R_{ins}, $R_{upn/upo}$, R_{pup}, R_{ups}).

7.1.2 Delta representation

The delta representation is the actual representation of the logical delta model which reflects the delta structures and the delta values. The generic structure of a "Delta" information item is shown in Figure 7.1:

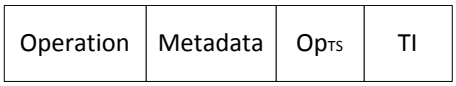


Figure 7.1: Delta information per column

OpType	Corresponding representations
1	Insertion
2	Update_old
3	Update_new
4	Upsert
5	Partial update

Table 7.1: Operation types

- The **Operation** column indicates how delta objects are generated. Values from 1 to 5 will be assigned to indicate the corresponding delta sets. The meaning of each number is illustrated in Table 7.1.
- The **Metadata** column contains the information of source table such as row key (Ide), column-family name (Cf) and column name (Col).
- The $\mathbf{Op}_{TS}$ column denotes when a specific data modification took place.
- The **TI** column is represented as $[Start, End)$ which denotes the valid temporal interval.

As described in Chapter 4, temporal data in a DW can be modeled using the tuple-timestmaping representation (TTR) or the explicit history representation (EHR). A TTR table appends each tuple with an explicit TI where EHR attaches a TI to each data version. When TTR is adopted as the temporal data model for DW, delta items have the same column-family names and column names as its corresponding source table with an additional TI column. Its row-key value of each delta item consists of *row key* from the source table, Op_{TS} and *Operation*. We call this temporal delta representation *row-level-representation*. However, when EHR is utilized, the row key of each delta item is composed by *Metadata*, Op_{TS} and *Operation*. To store the value of each delta item, an additional "Value" column is created. As TI is associated with each data version, we call this temporal delta representation *attribute-level-representation*.

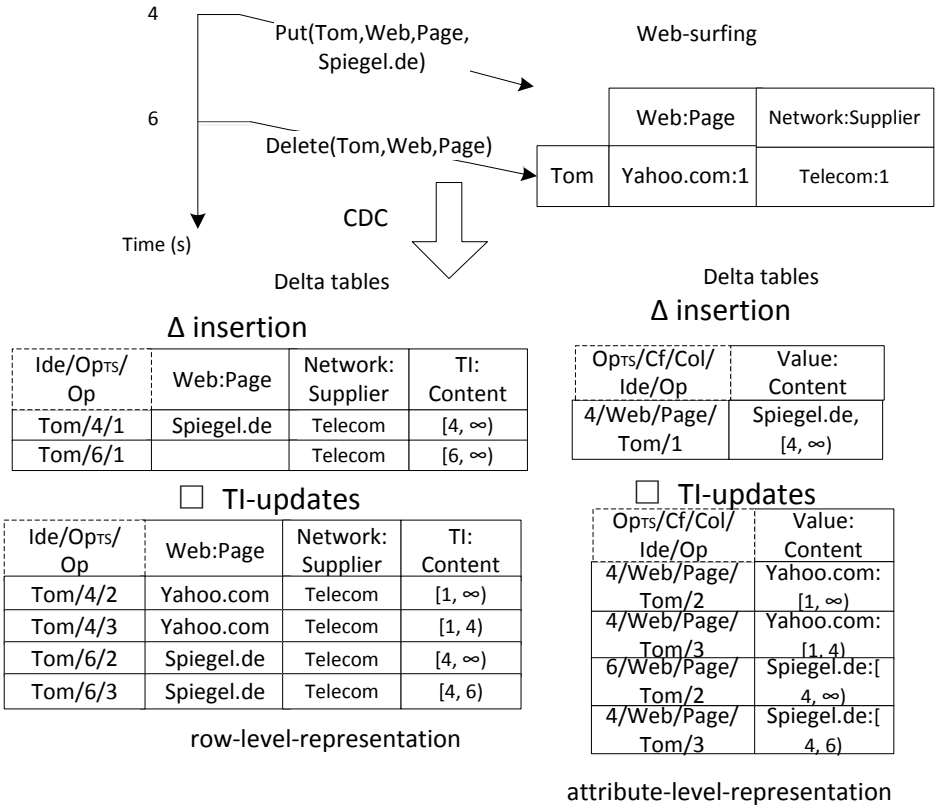


Figure 7.2: CDC examples

Figure 7.2 shows two examples of row-level-representation (RLR) and attribute-level-representation (ALR), respectively. The source table is called "Web-surfing" which contains two columns "Web: Page" and "Network: Supplier". In the example, there is one "Put" operation which generates a new data version for row key "Tom" and one "Column-Delete" operation which deletes all the data versions under "Web:Page" column for "Tom". The first "Put" operation is treated as an update as the row key "Tom" and the corresponding column have already existed. 3 tuples will be inserted into both RLR and ALR delta tables, namely, two for TI updates and one for insertion. For processing the deletion, 3 tuples will be again inserted into RLR where ALR only contains 2 tuples. For the ALR delta table, the deletion will only affect the TI of "Web:Page" column, and hence no changes will be recorded for "Network:Supplier" column.

Due to the previous example, it is obvious that the ALR maintains less redundant data than RLR. For example, string "Telecom" exists in every tuple of RLR delta table.

As we have integrated Op_{TS} which indicates when a specific data modification took place into the row key, it's very easy for an application to obtain a desired delta set by checking if the TSs of row keys are included in a specific time interval. Moreover, our Deltas table stores a historical delta set rather than the latest data changes. In consequence, the incremental applications and the data target (for example, DW) are capable to choose the desired parts to work with.

7.1.3 Enhanced delta representation for attribute-level

Although *attribute-level-representation* (ALR) maintains less data redundancy than *row-level-representation* (RLR), it leads to more data processing when a complete tuple is needed as output. In Figure 7.2, suppose we wish to select users who browse a specific web-site is less than 10 seconds (browsing time is calculated as $TI.End - TI.Start$). Initially, the content of this query is empty. At time point 4, user "Tom" leaves "Yahoo.com" and goes to "Spiegel.de". In consequence, delta item "Tom/4/3" (in RLR delta table) or delta item "4/Web/Page/Tom/3" (in ALR delta table) satisfies the selection criterion. The delta item "Tom/4/3" can be directly propagated to the DW. However, for delta item "4/Tom/Web/Page/3", the output of the incremental procedure contains only the partial row information as the delta representation is based on individual attribute.

Deltas table

TS/Cf/Col/ RK/Op	Value: Content	Web: Supplier
4/Web/Page/ Tom/2	Yahoo.com, [1, ∞)	Telecom, [1, ∞)
4/Web/Page/ Tom/3	Yahoo.com, [1, 4)	Telecom, [1, ∞)
4/Web/Page/ Tom/1	Spiegel.de, [4, ∞)	Telecom, [1, ∞)
6/Web/Page/ Tom/2	Spiegel.de, [4, ∞)	Telecom, [1, ∞)
4/Web/Page/ Tom/3	Spiegel.de, [4, 6)	Telecom, [1, ∞)

enhanced attribute-level-representation

Figure 7.3: Enhanced attribute-level-representation

To solve such issues, we can purse two options.

1. As the delta table maintains the history of data changes, the application can utilize the current state of the source table and the change data history to rebuild the state of a tuple at a specific time. For example, in Figure 7.2, after the delta satisfies the filter predicate, the application can use the row key derived from the delta object as a parameter for issuing a "*Get*" operation to the "Web-surfing" table to obtain the corresponding tuple value.

2. The other option is to include the unchanged data columns in the delta output. Compared to Figure 7.1, the unchanged data columns, namely the column names and the related values derived from the source table, are appended to the delta results. We call this delta representation "*enhanced attribute-level-representation*". Figure 7.3 shows an example of the enhanced delta representation.

   In the example, one column ("Network:Supplier") which denotes the unchanged column derived from source table ("Web-surfing") is appended to the Deltas table. Now the incremental application can directly generate the desired data output.

7.2 Change-data-capture approaches

In this section, we describe five feasible CDC approaches, namely, *Timestamp-based approach*, *Audit-column approach*, *Log-based approach*, *Trigger-based approach* and *Snapshot differential approach*, which are exploited to detect and extract deltas from WCSs. The Timestamp-based approach is a WCS-specific approach. Although the other approaches are well known for RDBMS or DW, their designs and implementations in the NoSQL context are restricted by the MapReduce programming model and the characteristics of the WCSs. Each CDC approach can only produce deltas at a certain level of completeness and a corresponding delta history due to its inherent characteristics, e.g. the Timestamp-based approach cannot generate inserted deltas and could only produce a partial delta history. Moreover, the limitations and pitfalls for each CDC approach will be discussed. For simplifying the descriptions, we chose to use attribute-level-representation as the output format for each CDC approach in our examples. More implementations details will be given in Section 7.3.

7.2.1 Timestamp-based approach

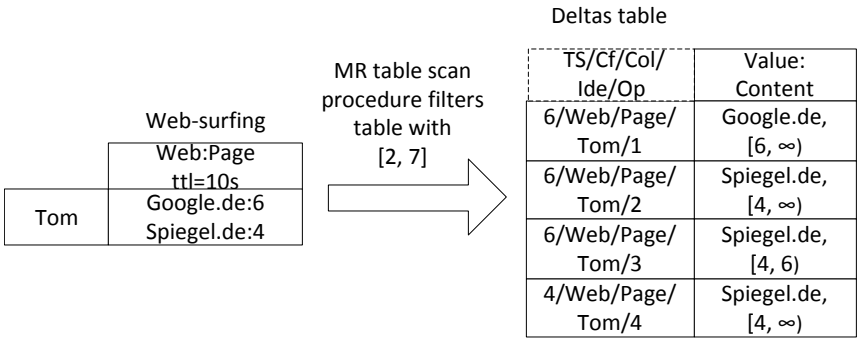


Figure 7.4: Timestamp-based approach

As we have already described in the previous chapters, each column of a WCS table stores multiple data versions. Besides the data value, each data version contains a TS which denotes when that data version was created by a "Put" command. The multiple data versions in a column can also be viewed

as a change data history for that column. For example, in Figure 7.4, the current data version for column "Web:Page" of row "Tom" is "Google.de:6" while the older data version "Spiegel.de:4" represents the data valid during [4, 6). Hence, it is feasible to scan a table with a certain time interval as a selection criterion to discover the delta candidates.

Figure 7.4 describes this approach. A MapReduce job is applied to scan the table with timing constraint [2, 7], which indicates to return the data values whose timestamps are larger than or equal to time point 2 and less than or equal to time point 7. The corresponding output is shown at the right-hand side of the figure. The data version "Google.de:6" is treated as update, as CDC can detect there already exists a data version, i.e. "Spiegel.de:4" before "Google.de" is inserted into the "Web:Page" column.

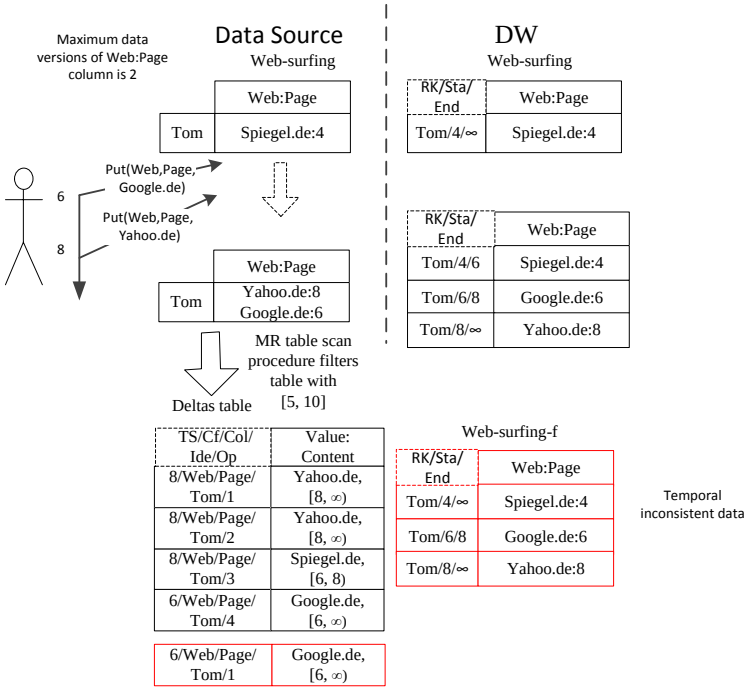


Figure 7.5: Detecting the type of the oldest data version

Notice that, "Spiegel.de:4" is classified as an upserted delta instead of an inserted delta. The reason is that each column in a WCS table cannot store infinite number of data versions. Take a look an example in Figure 7.5. Suppose the maximum data versions of column "Web:Page" is 2 and

we periodically copy the table values from data source to DW. If we propagate "(Google, [6, ∞)" as an insertion instead of an upsert. Web-surfing table (denoted as "Web-surfing-f") will contain temporal conflict data, e.g. "Tom/4/∞" and "Tom/6/8".

As there are no data versions older than "Spiegel.de:4", CDC cannot decide whether it is generated via insertion or update.

Although the Timestamp-based approach is easy and straightforward, there are several drawbacks. First, CDC cannot distinguish between update and insertion (e.g. "4/Web/Page/Tom/4"). Second, as deletions cannot be detected, the Deltas table can only contain a partial change data history.

7.2.2 Audit-column approach

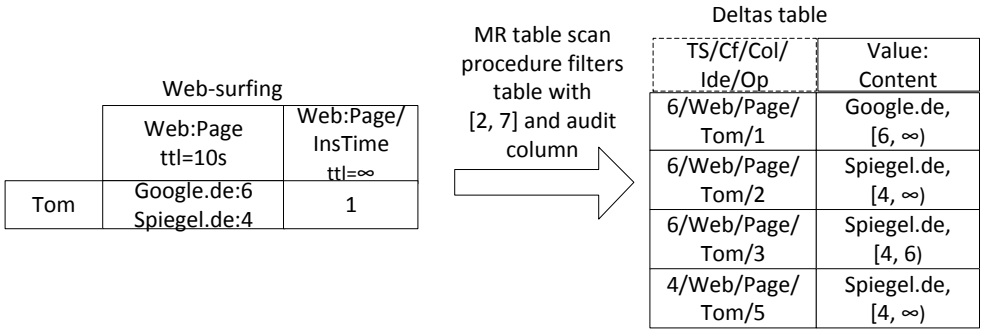


Figure 7.6: Audit-column approach

The Audit-column approach can be used to address some drawbacks of the Timestamp-based approach. Each source table is extended with auxiliary columns to record the data insertion time. Figure 7.6 depicts this approach. Now, "Spiegel.de:4" is categorized as R_{pup} (partial update) rather than R_{ups} (in Figure 7.4). Although this approach can differentiate update from insertion, deletions still cannot be detected and the delta history could be still partial. Moreover, unless such a column is already part of the database design for auditing process, the application logic needs to be added or changed to maintain the audit column.

7.2.3 Log-based approach

For recovery and durability reasons, each work node in WCSs maintains a WAL (Write-Ahead Log). A WAL is a form of redo log which requires all the data modifications applied to the database must be deferred until those changes have been recorded in the log file. When a crash happens, the database system can exploit the log entries to rebuild the state before the crash occurred. The content of the WAL can also be used to extract deltas.

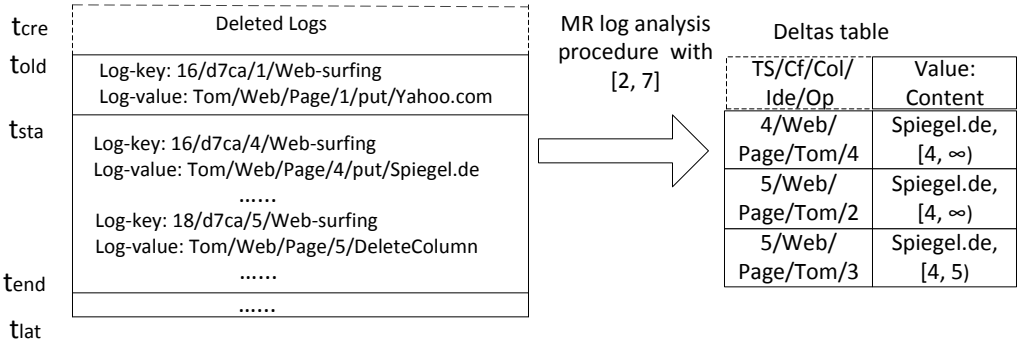


Figure 7.7: Log-based approach

The left hand side of Figure 7.7 shows a WAL segment for the "Web-surfing" table. Each log entry is organized as a key-value pair. The key part is composed of a log sequence number, node ID, log writing time and table name. The value part contains the information of the data modification. Different from RDBMS, a WAL in WCSs doesn't distinguish between update and insertion and only records "tombstones" for deletions.

Log entries may be automatically deleted by WCS when the data modifications have been made persistent on disk. Hence, without a specific configuration, CDC may only get a partial WAL for analysis. In Figure 7.7, we use t_{cre}, t_{old} and t_{lat} to denote the creation time of the log, the oldest and the latest timestamps among the existing log entries, respectively. [t_{sta}, t_{end}] indicates the time interval in which CDC wishes to capture deltas. A MapReduce log analysis procedure is applied to scan the partial WAL of the "Web-surfing" table with [$t_{sta} = 2$; $t_{end} = 7$]. Notice that log analysis procedure cannot decide whether "Spiegel.de:4" is generated via insertion or update, as it is the oldest log entry for row "Tom" under the "Web:Page"

column. The intuitive solution is to extend the scanning time interval from $[t_{sta}, t_{end}]$ to $[t_{old}, t_{end}]$. Nevertheless, as the WAL remains partial, it is still impossible to detect the inserted deltas. Hence, a full WAL needs to be saved if the complete delta sets are required. However, as WAL records the log information for every table residing on a cluster node, saving and analyzing the whole WAL will be space-consuming and time-consuming.

7.2.4 Trigger-based approach

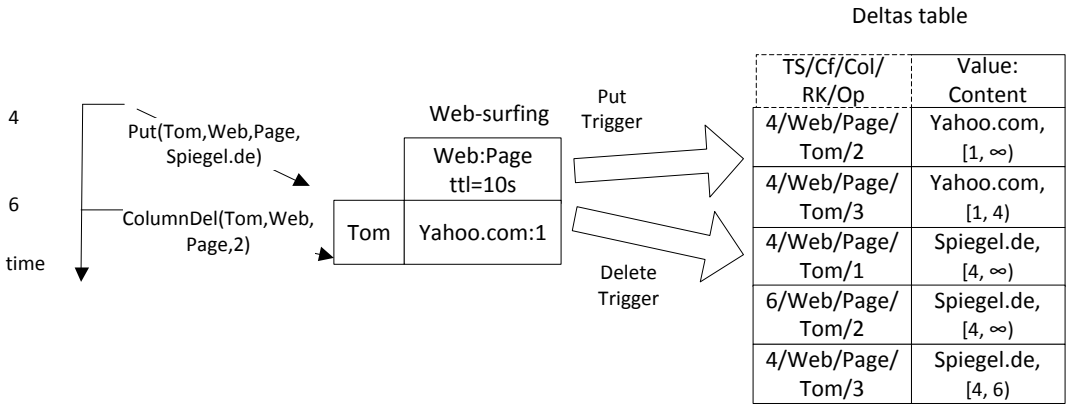


Figure 7.8: Trigger-based approach

A database trigger allows to automatically execute data manipulation language or other operations in response to certain events on a particular table. Hence, detecting and storing deltas can be easily implemented by triggers, if they are supported by the WCS.

Figure 7.8 shows an example of the Trigger-based approach. Two triggers are defined to monitor "Put" and "Delete" events on the "Web-surfing" table. After a "Put" command is performed, the "Put" trigger will first access the related column on the source table to check whether data versions already existed before the "Put" is executed. If yes, the new generated data version is considered as update. Otherwise, it is treated as insertion. For monitoring the "Delete" operation, the "Delete" trigger needs to first detect the deletion granularity, i.e. *data version*, *column* or *column family*. Then it accesses the source table to find the matching deleted tuples.

7.2.5 Snapshot differential approach

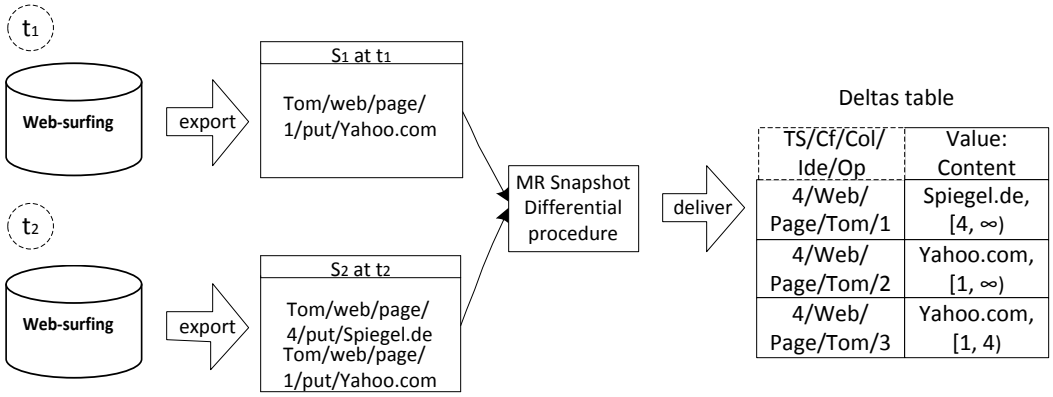


Figure 7.9: Snapshot differential approach

The Snapshot differential approach is typically utilized to capture deltas from legacy system or unsophisticated system. However, this approach can also be seen as a possible CDC candidate to detect and capture deltas from WCSs. Snapshot differential processing generally contains two phases:

1. Taking snapshots from data source at different timestamps.

2. Comparing snapshots to extract change data.

For backup purposes, most WCSs support commands (e.g. the "export" command of HBase) to dump valid content into the distributed file system. By "valid content", we mean data values which are not masked by tombstones or have expired.

Figure 7.9 describes this approach. First, two snapshots of the "Web-surfing" table are taken at t_1 and t_2, respectively. Then, two snapshot images are sent to the comparison procedure. In the snapshot comparison phase, all tuples which appear in S_2 but not in S_1 are considered as insertion deltas. In contrast, tuples which exist in S_1 but not in S_2 are treated as deletion deltas. Tuples which exist in both S_1 and S_2 but differ from each other are considered as update deltas. However, as the snapshot content doesn't contain any tombstone information, CDC cannot decide whether a data value

is deleted by "Delete" command or ttl expiration if t_2 is greater than the time of ttl expiration. When this situation occurs, we always assume the deleted data changes are generated via ttl expiration, namely, the End point of its TI is calculated by ttl. Moreover, it is possible that the Deltas table only represents a partial delta history, e.g. a data version which is inserted into the "Web-surfing" table after t_1, and is deleted due to ttl expiration or delete command before t_2 will not be captured.

7.3 Performance and implementation

In this section, we describe implementation details based on different delta representations, namely, row-level-representation, attribute-level representation and enhanced attribute-level representation. Moreover, we show performance charts to indicate the processing speed of each CDC approach. The experiments are performed on a six-node cluster (Xeon Quadcore CPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) running Hadoop (version 2.0.2-appending) and HBase (version 0.92), respectively.

Test data sets are generated through the "*workloada*" of YCSB (Yahoo Cloud Service Benchmark). At the "Loading" phase, 10 million tuples are inserted into a predefined "usertable" in HBase. The "usertable" contains one column family named "cf1" with 10 columns. The ttl property for "cf1" is set to 60 minutes and the data replication factor is 3. To avoid the "hot spot" issue (one node will receive a significant amount of data processing tasks while the other nodes are idle) in the cluster, the value of "*requestdistribution*" property is set to "uniform" which indicates that data are evenly distributed in the cluster. At the "Transaction" phase, we vary the proportion of data modifications applied to "usertable" through the properties of "*updateproportion*" and "*insertproportion*". As we can only indicate data modification based on update and insertion inside workloada, we write our own program to generate the delete operations.

7.3.1 Row-level-representation

The MapReduce framework is adopted as the application layer to implement the CDC approaches except for *Trigger-based* approach. The Time-based approach and the Audit-column approach are implemented only in the Map function by scanning the source table with certain time spans. Each TS in a

row indicates when the data modification, namely, update, upsert or partial update occurs. To maintain the audit columns, we define a "preCheck" trigger at the WCSs server side to monitor each "Put" command. When WCS receives a Put command, the "preCheck" will first check the corresponding columns. If no values exist, the insertion time for such a column is created.

For the Log-based approach, the whole WAL is maintained. To achieve that, we set the property "*hbase.master.logcleaner.ttl*" (used to define the life cycle of log) to a long time and incrementally copy the HBase log into a specific archive. To obtain the freshness of the "Deltas" table, we set the property "*hbase.regionserver.logroll.period*" to 1 min [1]. The log analysis procedure is implemented via Map and Reduce functions. The Map function reads the log entries as input and generates the intermediate key, which is the row key and the intermediate value, which consists of the column-family name, column name and the data value for each data modification. The Reduce function sorts the data value based on the TS in ascending order and treat the data values which belong to the oldest TS as insert delta where the other values will be treated as update or deletion. For implementing the Trigger-based approach, two *RegionObservers* [Hba] "*postPut*" and "*preDelete*" are defined. The RegionObserver provides hooks for data manipulation events, e.g. Put and Delete and is executed when certain events occurred. The "*postPut*" RegionObserver is executed after a Put command is finished and the "*preDelete*" RegionObserver is executed before executing a delete command. The functionality of these two RegionObservers have been described in Section 7.2.

We utilize two separate charts to depict the CDC performance for data generation and data deletion in Figure 7.10 and Figure 7.11 respectively, as the Timestamp-based approach and the Audit-column approach are not capable to detect deletions. The range of execution time of the five CDC approaches is between 3.7 to 60 minutes which consists of initializing MapReduce jobs, scanning WCS tables (or log files), CDC data processing and loading CDC results. Normally, we need 30s to 40s to initialize MapRe-

[1] The value of log-roll period can be configured at the granularity of milli-second. However, the shorter the period, the smaller the log files that will be produced. For example, if the period is set to 1 second, HBase will generate 395 log files (from 2M to 5M) when inserting one million rows. In contrast, if the period is set to 1 minute, HBase will generate 130 log files (from 8M to 19M). Although the short log-roll period can guarantee the delta freshness, MapReducek is not suitable to process a large amount of small files. Hence, there is a trade-off between the delta freshness and the performance of MapReduce jobs.

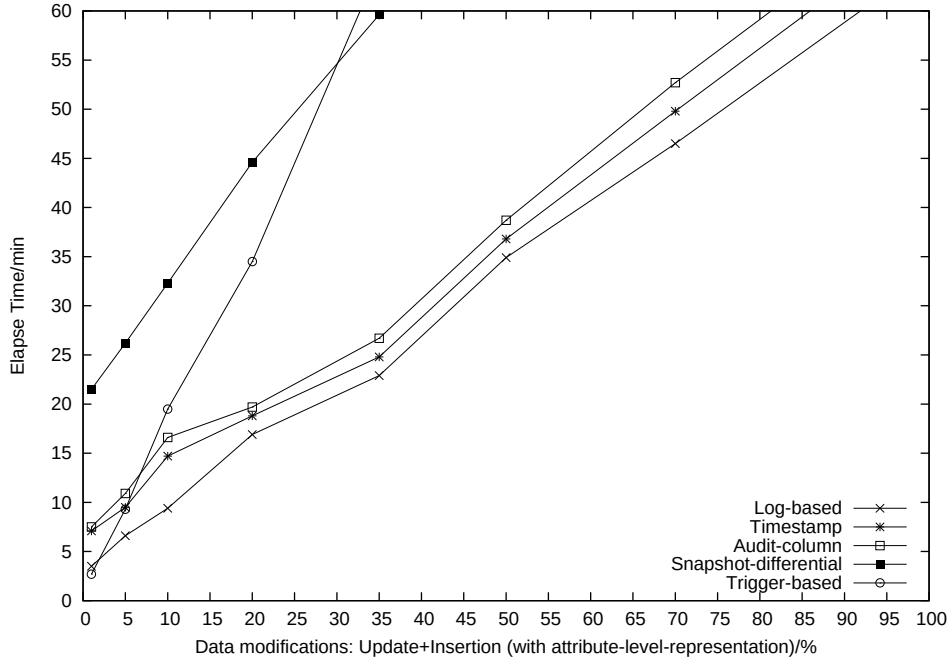


Figure 7.10: Update+Insertion (row-level-representation)

duce jobs in the cluster. The low performance of scanning HBase tables via MapReduce have the following causes:

1. Even if data locality is guaranteed, local reads in HBase are handled the same way as remote reads.

2. As valid data versions of one column could be distributed across multiple HFiles (one HFile consists of a set of key-value pairs and it is a basic storage unit for HBase), merging HFiles through table scan will take a long time.

3. HBase does not support data prefetch mechanisms when utilizing MapReduce.

4. It is possible that the scan performed by a $TaskTracker$ (a MapReduce slave node which executes Map and Reduce function) would like to access data hosted on other nodes. Hence, data shipping is needed to process the table scan.

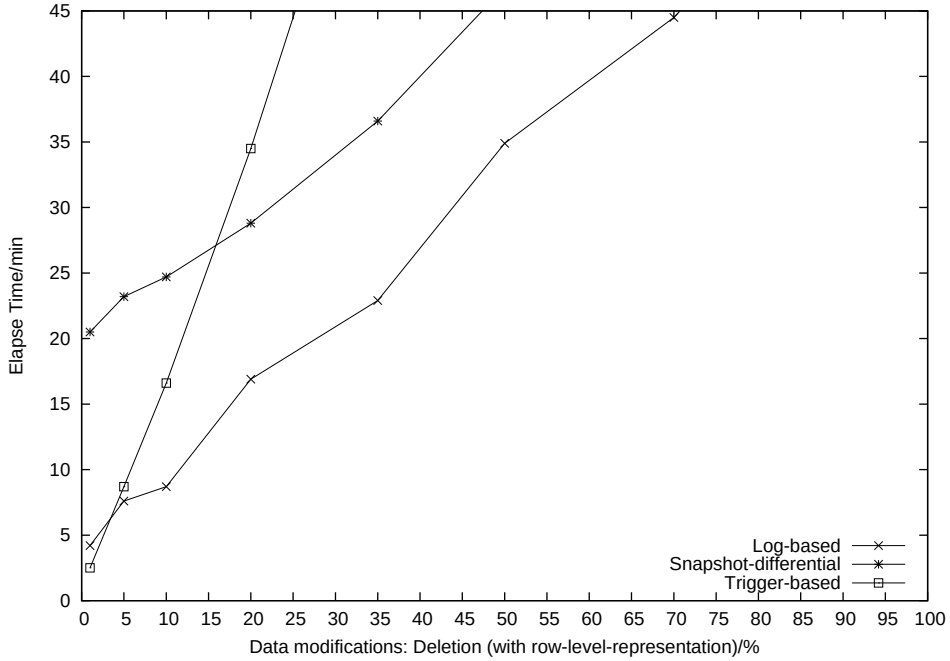


Figure 7.11: Deletion (row-level-representation)

As HBase is memory-intensive, when Garbage Collection is executed by the JVM (Java Virtual Machine) during loading CDC results, any client requests are stalled and the server will pause for a significant amount of time [Hba].

In both charts, the Trigger-based approach shows the best performance if the fraction of change data is small. The reason is that the triggers do not cache any table locations. Hence, triggers have to find the correlated working regions for every invocation. In Figure 7.10, the Audit-column approach is slower than the Timestamp-based approach as it needs to process the audit columns. For both figures, it is at first surprising to see that the Log-based approach shows a better performance than the other approaches for the middle and high percentages of data modifications, despite the fact that it needs to process more data (in our experiment, WAL contains the log information for a "usertable" table and a "Deltas" table). The reason is that reading data from HDFS is approximately 5 times [Hba] (in our experiment, the ratio is 2.3) faster than reading the same amount of data from HBase when utilizing the MapReduce framework. Moreover, the header of each

WAL fragment maintains metadata, such as table name and log creation time. Hence, the log analysis procedure can filter unnecessary log entries very rapidly. The Snapshot differential approach shows a bad performance compared to the Log-based approach, as it needs to scan the whole table to produce a new snapshot and one MapReduce snapshot comparison needs to be executed.

7.3.2 Attribute-level-representation

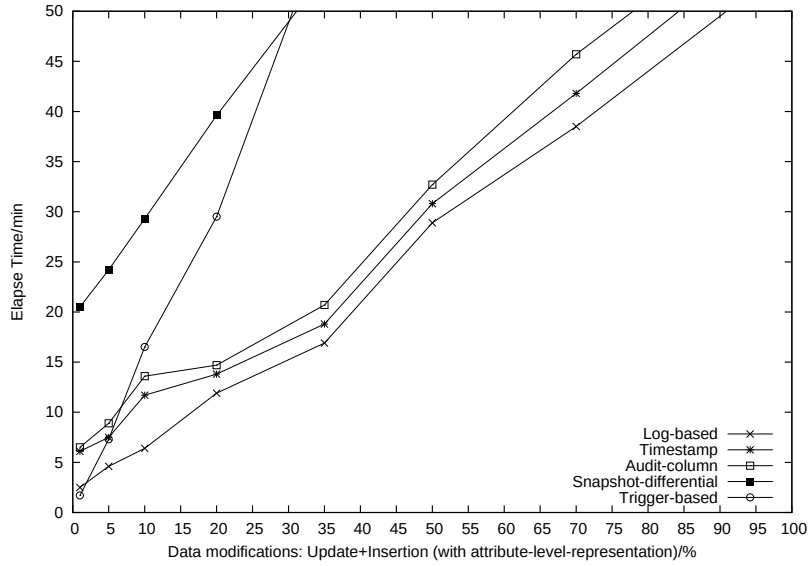


Figure 7.12: Update+Insertion (attribute-level-representation)

Different from the row-level-representation, the attribute-level-representation stores deltas based on each individual column. To implement the Timestamp-based approach and the Audit-column approach, the TSs of each individual column will be considered as time points when data modifications occur. In contrast to the row-level-representation, the access of the other columns to extract the complete row (in which its column values are valid during the same period) is not needed. When the Reduce phase is needed (for the Log-based approach and Snapshot differential approach), the intermediate key of Map function is composed by row key, column-family name

and column name, and the intermediate value consists of the data value for each data modification.

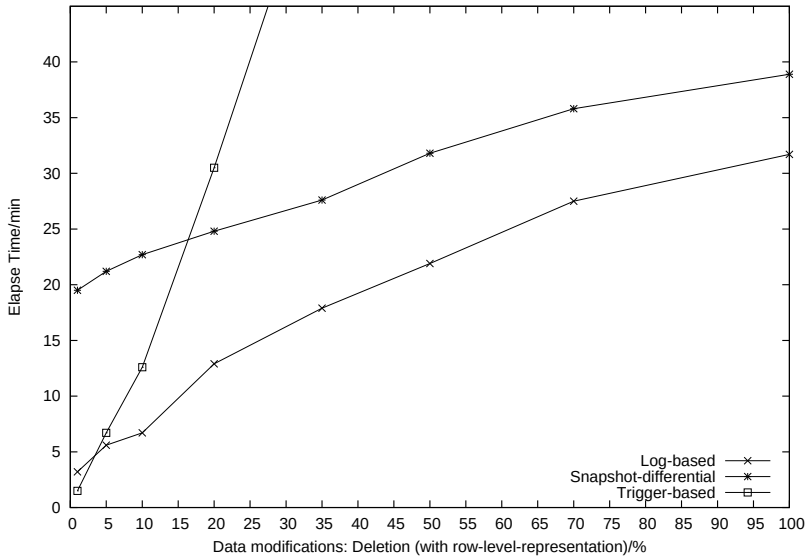


Figure 7.13: Deletion (attribute-level-representation)

Again, we separate the performance charts into two. Figure 7.12 shows the performance for updated and inserted data and Figure 7.13 depicts the deleted data. The delta extraction time ranges between 1.8 to 50 minutes. Clearly, generating the attribute-level-representation is faster than producing the row-level-representation, as the attribute-level-represen-tation maintains less redundant data.

7.3.3 Enhanced attribute-level-representation

In contrast to the attribute-level-representation, each CDC approach has to generate the corresponding unchanged data columns in the enhanced attribute-level-representation. For implementing the Timestamp-based approach and the Audit-column approach, we need to use the temporal interval (TI) of each changed data version as a selection criterion to retrieve the related unchanged data versions from the other columns. For implementing the Log-based approach, the output key of the Map function is the row key itself rather than the combination of row key and column-family name. The

Reduce function will rebuild the row, detect the changed data and find the corresponding data versions of unchanged columns.

The Snapshot differential approach is also implemented through Map and Reduce functions. The Map function reads two snapshots as inputs and the data processing logic of the Reduce function is the same as for the Log-based approach. For implementing the Trigger-based approach, as the trigger will issue a "Get" command to the data source to obtain the row information when a "Put" or "Delete" command occurs, it can directly obtain the data versions from unchanged columns without additional comparison processing.

Again, we separate the performance charts into two. Figure 7.14 shows the performance for updated and inserted data and Figure 7.15 depicts the deleted data. The delta extraction time ranges from 2.2 to 60 minutes. The data processing time consists of initializing MapReduce jobs, detection of changed data, extraction of unchanged data and delta loading. We can notice that extracting unchanged data has heavily influenced the performance compared to Figure 7.12 and Figure 7.13.

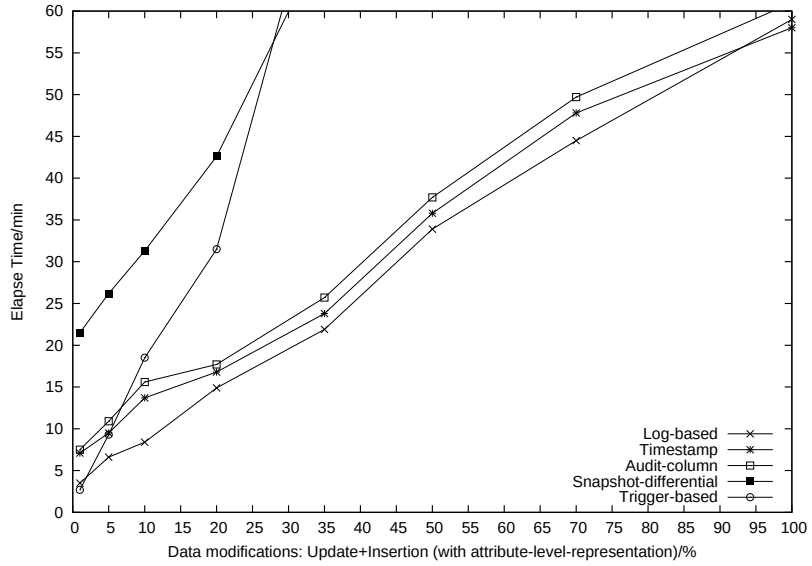


Figure 7.14: Update+Insertion (enhanced attribute-level-representation)

In our experiment, the source table contains one column family with 10 columns. Hence, to generate the corresponding unchanged columns for every

delta object, the CDC has to use the TS of that delta as a selection criterion to retrieve the remaining 9 columns.

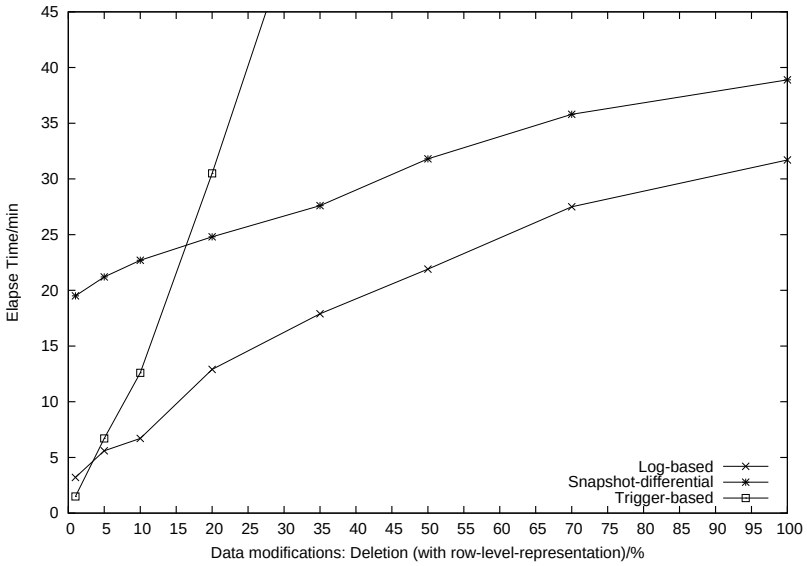


Figure 7.15: Deletion (enhanced attribute-level-representation)

For both charts, the Trigger-based approach is still the best one when the data modification rate is low. Different from the Figure 7.12, the Log-based approach is not the fastest approach any more for generating inserted and updated deltas. The Timestamp-based approach is faster than the Log-based approach when the data modification rate is high. The reason is that after the Map function accomplishes its task, the Reduce function needs to build the row information from multiple entries as each log entry only records the data modification at the data version level. When the number of data modifications increases, the Reduce function needs a long time to build the rows. For capturing deleted deltas, the Log-based approach is still the best when the data modification rate is high.

7.4 Summary

Change-data capture (CDC) is a prerequisite data processing task of the incremental maintenance of data warehouses or data transformation results.

In this chapter, we described our generic logical delta model and the corresponding delta representations. Moreover, we discussed how incremental applications can utilize deltas to refresh the data target. Due to the various temporal delta models utilized in WCSs, each delta item can be represented using row-level-representation or attribute-level-representation. When the attribute-level-representation is used, a tuple reconstruction is needed for certain incremental applications, e.g. a filter operation. To achieve that, we could either let the applications rebuild the row or append the unchanged data columns to the delta output (enhanced attribute-level-representation). The drawback of the first strategy is that it increases the processing overhead of incremental applications. Although the second approach facilitates the data processing task, it increases the delta extraction time and disk usage.

To extract temporal deltas from WCSs, five CDC approaches, namely, *Timestamp-based approach*, *Audit-column approach*, *Log-based approach*, *Trigger-based approach* and *Snapshot differential approach*, can be used.

Based on the discussions made in Section 7.2, we notice that the Timestamp-based approach is the most convenient approach by only scanning the source table with the denoted time spans. However, this approach cannot distinguish between insertion and update. The Audit-column approach is an improvement of the Timestamp-based approach but auxiliary columns need to be created to record the data insertion time. The problems with these two approaches are: First, both lack the ability to extract delete deltas. Second, a full table scan will be very inefficient if only a small portion of the source table is modified. In the Log-based approach, the completeness of deltas depends on the completeness of the WAL (write-ahead log). However, storing and analyzing the whole WAL is space-consuming and time-consuming. When a WCS provides active capabilities such as triggers, the Trigger-based approach can be adopted as an alternative to the Log-based approach. The Snapshot differential approach is usually seen as "a last resort" for capturing deltas. As the volume of source data grows, more and more data has to be extracted and larger and larger comparisons have to be performed.

We summarize the kinds of deltas which can be extracted by our CDC approaches in Figure 7.16 . The first column indicates the name of each CDC approach and the remaining columns denote the different delta sets. A check mark (✓) in the table denotes the CDC approach has the ability to extract the corresponding delta set. For the "Delta His" column, "P" and "C" denote partial and complete delta history, respectively.

For comparing the performance of different CDC approaches, we consid-

CDC approaches	R_{ins}	$R_{upo/upn}$	R_{ups}	R_{pup}	Delta his
Timestamp-based approach		✓	✓		P
Audit-column approach	✓	✓		✓	P
Log-based approach (partial)		✓	✓		P
Log-based approach (complete)	✓	✓			C
Trigger-based approach	✓	✓			C
Snapshot differential approach	✓	✓			P

Figure 7.16: Summarization of delta sets for various CDC approaches

ered three different delta formats. For all three, the Trigger-based approach is always the best option when the data modification rate is low. When the data modification rate is high and the CDC uses the attribute-level-representation, the Log-based approach is faster than the other approaches. However, if the CDC utilizes the row-level-representation and enhanced attribute-level-representation, the Timestamp-based approach is faster than the Log-based approach for generating inserted and updated deltas. Nevertheless, even though the Log-based approach is slightly slower at a high data modification rate, it is still preferable as it can generate a complete delta history.

Chapter 8

Incremental temporal data recomputation with complete temporal deltas

In Chapter 4 and Chapter 5, we have introduced how temporal data can be modeled and processed in WCSs, namely, the *tuple-timestamp model* which is processed by the *TTRO* temporal operator model and the *explicit history representation* which is handled by the *CTO* temporal operator model.

Before focusing on incremental propagation of temporal deltas based on these two temporal data models (and their corresponding operators), we described in Chapter 7 how temporal change data (deltas) can be modeled and introduced their corresponding physical representations, namely, *row-level-representation*, *attribute-level-representation* and *enhanced attribute-level- representation*. According to our previous discussions, not every change-data capture (CDC) approach is able to generate complete delta sets, i.e. insert and delete delta sets where update is represented as a pair of deletion and insertion.

In general, incremental temporal recomputation can be achieved in two different ways:

1. Transform temporal data (including both temporal base data and change data) and temporal queries into their non-temporal counterparts and reuse the view maintenance approaches for non-temporal data [GM95].

2. Directly propagate temporal deltas based on the temporal operators. Generally, temporal deltas are classified as insertion and deletion where

the update is represented as a deletion/insertion pair. As already described in Section 6.3.2, [YW98] defined a set of temporal delta propagation rules based on the tuple-timestamping model and the temporal relational algebra. Temporal queries are classified into different categories according to their semantics and corresponding temporal deltas are generated. The temporal delta propagation rules look the same as their non-temporal counterparts [QW91].

Although the first approach can be seamlessly integrated with existing database systems, e.g. relational database systems, transforming temporal data can heavily decrease query performance [DBG12]. Moreover, transformations between temporal and non-temporal queries are non-trivial. In consequence, for incrementally recomputating temporal query results, we adopt the second approach.

In this chapter, we introduce how to incrementally propagate the complete temporal deltas based on the TTRO and CTO models. To achieve that, several challenges arise.

- When propagating temporal data changes (deltas), the traditional way is to treat an update as a deletion/insertion pair. However, as a WCS usually maintains a large number of columns, such an approach is inefficient when only a small number of columns are changed. Hence, a better way is to treat the update as an individual case.

- As a DW often maintains the complete source data history, no current data in base relations (see DW architecture in Section 5.1) will be deleted or overwritten. Hence, the extracted deltas from data sources are represented as insertions and TI-updates. However, during temporal delta propagation, a TI-update can cause a deletion. For example, the before-state of a TI-update tuple satisfies the temporal predicate but its after-state not. Consequently, the temporal delta propagation is not *closed*. We say a delta propagation is *closed* when the input and output of that delta propagation have the same delta types, i.e. no new delta types are generated during delta propagation.

- For the explicit history representation and CTO operator model, as temporal deltas are represented at attribute-level, a set of new temporal propagation rules need to be defined.

- According to Section 6.3.2, the incremental recomputation of temporal aggregations can be achieved using different types of indices, e.g. "aggregation-tree" [KS95] or "SB-tree" [YW03]. Although the previous approaches can be used by WCSs, new data structures and functionality are needed. Moreover, as data in the WCS is normally distributed in a cluster, maintaining a distributed index (e.g. B-tree in a distributed fashion) is time-consuming and non-trivial.

The rest of sections addresses the issues mentioned above. We first introduce a new representation of the complete temporal delta set and describe how temporal queries can be classified. Then, we describe how temporal delta propagation rules can be defined based on *TTRO* and *CTO* operators, respectively.

8.1 New complete temporal delta representation and temporal query classifications

For maintaining temporal views [YW98], temporal deltas are propagated based on deletions ∇ and insertions Δ, with updates represented as deletion/insertion pairs. However, as a WCS table usually contains a large number of columns, this strategy can be inefficient when only a small number of columns are changed/updated. The more cost-effective way is to explicitly introduce update deltas in the delta representation model.

As already described in Section 7, we represent the complete temporal deltas extracted from data sources as insertions Δ and TI-updates $\square$. $\square$ is represented as a pair which contains the before $\square^-$ and after state $\square^+$ of the TI-update tuples. However, during temporal delta propagations, $\square$ can cause deletions ∇ which are not defined in our complete temporal delta sets. For example, the before-state $\square^-$ satisfies the predicate and the after-state $\square^+$ not. In consequence, the temporal delta propagation is not *closed*, namely, a new delta type is generated through delta propagation. Figure 8.1 shows this example. Suppose we have the following query.

Query 1. *Select Network suppliers whose speed are greater than or equal to 1000K and are used currently by users*:
$\sigma^T_{N.Speed \geq 1000K \wedge TI.Start \leq NOW < TI.End}(N)$. N is an alias of "Network". Symbol NOW represents the current time.

In Figure 8.1, the tuple-timestamping model is used. "Network" is a table to record Internet information about users and temporal interval (TI) denotes how long a user uses the service of a specific Internet supplier. The initial state of "Network" is described at the left-top. V_1 denotes the initial state of the temporal view. Δ and $\square$ represent deltas which are generated by an update that modifies column "Network:Speed" at time point 10. When propagating those temporal deltas, $\square^+$ and Δ do not satisfy the filter predicate but $\square^-$ does. In consequence, tuple "Tom/5/∞" in V_1 needs to be deleted (V_3 represents the new state of view). In this situation, a new delta type *deletion* ∇ is generated.

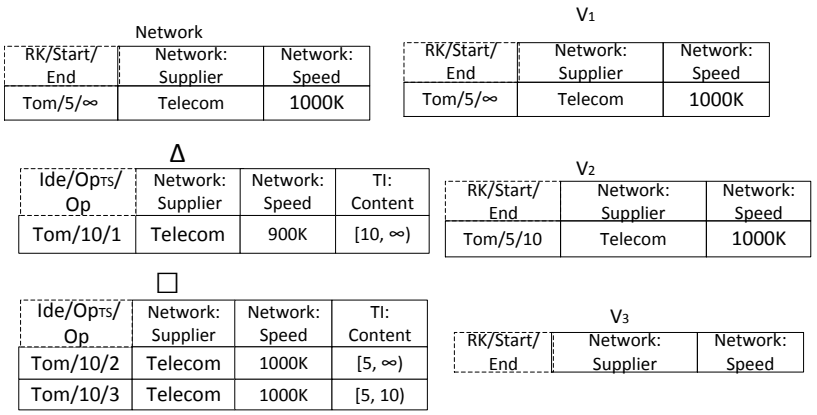


Figure 8.1: New generated delta types

Nevertheless, if we modify the previous query to $\sigma^T_{N.Speed \geq 1000K}(N)$, $\square$ will not cause tuple deletion. Instead, TI-update can be directly propagated. Corresponding query result is shown in V_2.

Due to the above example, we can notice that $\square$ can be handled as an individual class when temporal query does not have the explicit references to time [YW98]. Otherwise, a new delta type ∇ can be generated. Analogy to [YW98], we classify the temporal queries to *Snapshot-reducible* [SAea94] and *Extended Snapshot-reducible* [DBG12]

Definition 8.1.1 (Snapshot Reducibility). Let $r_1, ..., r_n$ be temporal relations, q^t a temporal query and q the corresponding non-temporal query, $TI = r_1.TI \cup ... \cup r_n.TI$, $S_p(r)$ the timeslice operator. Query q^t is snapshot-reducible iff $\forall t_i \in TI | S_{t_i}(q^t(r_1, ..., r_n)) = q(S_{t_i}(r_1)..., S_{t_i}(r_n))$.

In general, snapshot reducibility does not apply to temporal queries with predicates which explicitly reference to time (since TI is removed by $S_p(r)$

[SAea94]). To overcome such issue, we can propagate TI as non-temporal attributes and attach it to each snapshot [DBG12]. To generate an extended snapshot, we define a new operator $\kappa_p(r)$ where $\kappa_p(r) = (S_p(r), r.TI)$. The definition of extended snapshot reducibility is given as follows.

Definition 8.1.2 (Extended Snapshot Reducibility). Let $r_1, ..., r_n$ be temporal relations, q^t a temporal query and q the corresponding non-temporal query, $TI = r_1.TI \cup ... \cup r_n.TI$. Query q^t is extended snapshot-reducible iff $\forall t_i \in TI | \kappa_{t_i}(q^t(r_1, ..., r_n)) = q(\kappa_{t_i}(r_1)..., \kappa_{t_i}(r_n))$. When transforming q^t to q, additional non-temporal predicates need to be specified in q. For example, when transforming temporal join predicates, an explicit TI-overlapping constraint needs to be added in the non-temporal predicate.

Lemma 8.1.3. *Let $q(r)$ be a temporal query. Δ and $\square$ represent insert and TI-update deltas extracted from r.*

- *$\square$ can be propagated as update if q is snapshot-reducible.*
- *$\square$ has to be propagated as deletion/insertion pairs when q is extended snapshot-reducible.*

Proof. When q is snapshot-reducible, only non-temporal comparisons exist in q. As $\square$ represents TI-updates, it can only affect the temporal view if the non-temporal attributes of $\square$ satisfy the predicates in q. Hence, $\square$ can be propagated as update. When q is extended snapshot-reducible, $\square$ can generate deleted deltas during delta propagation(see example in Figure 8.1). Consequently, the temporal delta propagation is not *closed*, namely, the temporal delta propagation generates a new delta type which is not defined in its input. Hence, $\square$ should be represented and propagated as deletion/insertion pairs. $\square$

For the extended snapshot reducible queries, as symbol NOW can be included in the predicate, the temporal predicate varies from time to time. In consequence, not every extended snapshot reducible query can be incrementally recomputed. Figure 8.2 shows this example. Suppose we have the following query.

Query 2. *Select all new users within the past 2 days*:
$\sigma^T_{NOW-2 \leq TI.Start \leq NOW}(N)$. N is an alias of "Network". Symbol NOW represents the current time.

In Figure 8.2, V_6 and V_8 represents the query results which are executed at day 6 and day 8, respectively. At day 6, interval $[NOW - 2, NOW]$ is evaluated as [4, 6]. In consequence, tuple "Jim/5/∞" is included in V_6. When executing temporal query at day 8, as value of $[NOW - 2, NOW]$ is changed to [6, 8], no tuples in Network table satisfy the temporal predicate and V_8 is empty. However, it is impossible to use the incremental recomputation strategy to refresh the results of Query 2 namely, from V_6 to V_8, as no data changes are made to the base relation ("Network"). To obtain the current state of the query results, a full recomputation needs to be performed.

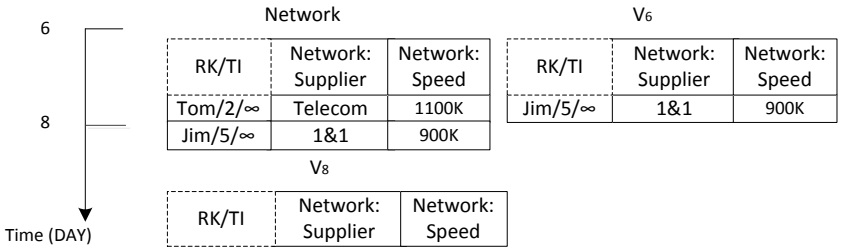


Figure 8.2: Temporal query which cannot be incrementally recomputed

Due to the previous example, the extended snapshot reducible queries are further classified based on whether they are *idempotent* or not. *Idempotence* is a property which describes that with the same input, a function can be applied multiple times (at different time points) without changing the result. For example, the number 1 is an idempotent of multiplication $\times$: $1^2 = 1^3 = 1^4 = 1$. We use notation $E(q, r, t)$ to represent the result of temporal query q which is executed against table r at time t. We say q is idempotent if and only if with same r, $E(q, r, t_i) = E(q, r, t_j)$, where $t_i \neq t_j$, namely, no matter when q is executed, its result is always the same.

Lemma 8.1.4. *Let $q(r)$ be an extended snapshot reducible query. The result of $q(r)$ can be incrementally maintained when q is idempotent.*

Proof. When $q(r)$ is not idempotent, its result can be changed without modifying r. In consequence, to obtain the current state of $q(r)$, only full recomputation is feasible. □

To use Lemma 8.1.4, we can detect that Query 2 is not idempotent, as $V_6 \neq V_8$. In consequence, it cannot be incrementally maintained.

In the following sections, we give the temporal propagation rules based on the snapshot-reducible and extended snapshot-reducible queries, respectively.

Notice that, when the extended snapshot-reducible queries are used in the following sections, we mean the extended snapshot-reducible queries which are idempotent.

8.2 Incremental temporal data recomputation with TTRO operators

The TTRO operator model is used when temporal data is modeled using tuple-timestamping. It is an extension of temporal relational algebra with minor modifications. In this section, we introduce how incremental temporal propagation rules can be defined based on Project π^T_A, Filter σ^T_p, Join $\bowtie^T_p$ and Group-by $f_{A'}(\gamma^T_{(A,TG,W)})$ with the snapshot-reducible and extended snapshot-reducible queries.

8.2.1 Snapshot-reducible queries

The characteristics of snapshot-reducible queries is that temporal query does not have the explicit reference to time and TI-updates $\square$ can be directly propagated (see Lemma 8.1.3).

Project π^T_A

π^T_A returns columns which are denoted as projection attributes A. Based on the data model requirements of WCSs, the row key column must be included in the final results even if it is not specified as a desired projection attribute. In consequence, π^T_A satisfies the *key-preserving* property. The change data propagation rules of π^T_A are described in Table 8.1.

Delta_type	**Modifications**
Δ	$\pi^T_A(\Delta)$
$\square$	$\pi^T_A(\square)$

Table 8.1: Incremental procedure for π^T_A

In Table 8.1, the left side represents the delta type and the right side denotes the corresponding output. To apply modifications to the materialized π^T_A result, insertions and TI-updates will be performed.

Filter σ_p^T

Delta_type	**Modifications**
Δ	$\sigma_p^T(\Delta)$
$\square$	$\sigma_p^T(\square)$

Table 8.2: Incremental procedure for σ_p^T

σ_p^T selects the tuples which satisfy the filter predicate and discards all others. The selection predicate p has a form, such as $p_1 \circ ... \circ p_n$ where each p_i $(1 \leq i \leq n)$ is a non-temporal comparison and $\circ$ represents the logical connectives, such as $\wedge$, $\vee$ or $\neg$. The incremental procedure for σ_p^T is described in Table 8.2.

Join $\bowtie_p^T$

S^T	R^T	**Modifications**
S_0	R_0	
S_0	Δ_R	$S_0 \bowtie_p^T \Delta_R$
S_0	$\square_R$	$S_0 \bowtie_p^T \square_R$
Δ_S	R_0	$\Delta_S \bowtie_p^T R_0$
$\square_S$	R_0	$\square_S \bowtie_p^T R_0$
Δ_S	Δ_R	$\Delta_S \bowtie_p^T \Delta_R$
Δ_S	$\square_R$	$\Delta_S \bowtie_p^T \square_R^+$
$\square_S$	Δ_R	$\square_S^+ \bowtie_p^T \Delta_R$
$\square_S$	$\square_R$	$\square_S \bowtie_p^T \square_R$

Table 8.3: Incremental procedure for $\bowtie_p^T$

When joining two temporal tables S^T and R^T, tuples in the result table are generated when two joining tuples satisfy the join predicate and are valid during the same period of time. We decompose two join tables as R_0, Δ_R, $\square_R$, S_0, Δ_S and $\square_S$ where R_0 and S_0 represent the unchanged data of R and S, respectively. The incremental procedure for $\bowtie_p^T$ is described in Table 8.3. When propagating Δ_R or Δ_S, the types of modification required on the join result table are always insertions (rows 2, 4 and 6-8). A TI-update will be applied to the join result when the data modification types of both join operands are TI-update (row 9) or one is TI-update and the other is unchanged (rows 3 and 5).

We use an example in Figure 8.3 to illustrate how the above procedures can be used. Suppose we have $N \bowtie_{N.Supplier=C.RK}^T C$, where N and C denote

"Network" and "Company", respectively. Q_o represents the initial join result. To calculate the updates for Q_o,

- $\Delta_n \bowtie^c \Delta_c = \emptyset$.
- $\Delta_n \bowtie^c \square_c^+ = \emptyset$.
- $\square_n^+ \bowtie^c \Delta_c = \Delta_v$. An inserted tuple "Tom/1&1/9/1" is generated.
- $\square_n \bowtie^c \square_c = \square_v$. The TI of tuple "Tom/1&1" will be modified from $[5, \infty)$ to $[5, 9)$.

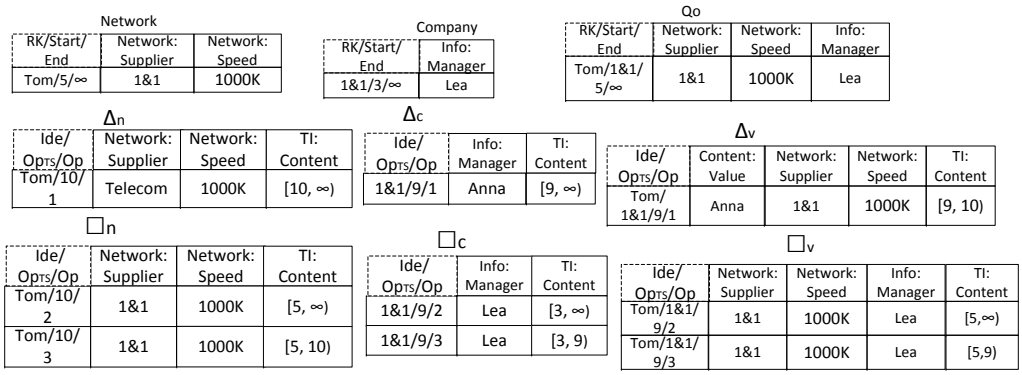


Figure 8.3: Example for incremental join procedures

8.2.2 Extended snapshot-reducible queries

When temporal query q is extended snapshot-reducible, q has an explicit reference to time, i.e. predicates in q have temporal comparisons. In consequence, $\square$ can cause deletions and temporal delta propagations are not closed. To overcome such issue, we represent the extracted deltas as deletions ∇ and insertions Δ. Delta transformation between $(\Delta, \square)$ and (Δ, ∇) is given as follows:

- $\Delta = \Delta \cup \square^+$.
- $\nabla = \square^-$.

In the following, we give the algorithms for incrementally maintaining temporal views based on ∇ and Δ.

Project π_A^T

The incremental procedures for π_A^c operator is described in Table 8.4.

Delta_type	**Modifications**
Δ	$\pi_A^T(\Delta)$
∇	$\pi_A^T(\nabla)$

Table 8.4: Incremental procedure for π_A^T

Filter σ_p^T

The incremental procedures for π_A^c operator is described in Table 8.5.

Delta_type	**Modifications**
Δ	$\sigma_p^T(\Delta)$
∇	$\sigma_p^T(\nabla)$

Table 8.5: Incremental procedure for σ_p^T

Join $\bowtie_p^T$

The incremental procedure for $R \bowtie_p^c S$ is described in Table 8.6.

S^T	R^T	**Modifications**
S_0	R_0	
S_0	Δ_R	$S_0 \bowtie_p^T \Delta_R$
S_0	∇_R	$S_0 \bowtie_p^T \nabla_R$
Δ_S	R_0	$R_0 \bowtie_p^T \Delta_S$
∇_S	R_0	$\nabla_S \bowtie_p^T R_0$
Δ_S	Δ_R	$\Delta_S \bowtie_p^T \Delta_R$
Δ_S	∇_R	
∇_S	∇_R	$\nabla_S \bowtie_p^T \nabla_R$
∇_S	Δ_R	

Table 8.6: Incremental procedure for $\bowtie_p^T$

8.2.3 Working together

In this section, we give an example to combine various incremental procedures together. Suppose we have two tables "Network" (N) and "Company" (C) (See Figure 8.4) and the following temporal view definition:
$V = \sigma^T_{N.Speed<1000 \wedge duration(TI)<10}(N) \bowtie^T_{N.Supplier=C.RK} C$.

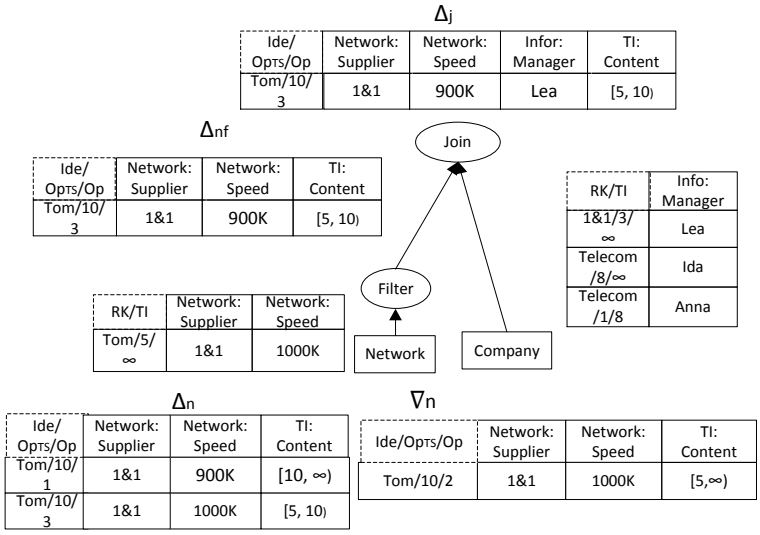


Figure 8.4: Example of temporal view maintenance with ATM

The abstract syntax tree and the corresponding data (including base relations and temporal deltas) are shown in Figure 8.4. As the view definition contains a temporal comparison ($duration(TI)$), it is classified as an extended snapshot-reducible query. Hence, temporal deltas are represented as ∇ and Δ. Initially, V is empty. At time point 10, 1000K is updated to 900K. For the filter operation, delta item "Tom/10/3" satisfies the predicate (result is denoted as Δ_{nf}). For the temporal join operation, table "Company" is viewed as unchanged data, i.e. C_0. We calculate $\Delta_{nf} \bowtie^T C_0$ and Δ_j is generated.

8.2.4 Temporal aggregation $f_{A'}(\gamma^T_{(A,TG,W)})$

In the temporal context, data can be grouped based on non-temporal attributes, temporal attributes (temporal instance or temporal interval) or both. To represent non-temporal aggregation, TG and W are not set (or

set to *null*). To specify *Instantaneous temporal aggregation* (ITA), TG is set to the finest granularity of the temporal table and W is set to 0. When specifying *cumulative temporal aggregation* (CTA), W is set to a positive integer.

In the non-temporal database context, view maintenance with aggregation functions has been studied for a long time, see [GM95]. In the temporal context, several approaches [KS95, YW03] addressed the incremental recomputations based on ITA and CTA. Generally, temporal deltas are classified as deletion ∇ and insertion Δ where an update is represented as a deletion/insertion pair. To implement the incremental recomputation, new index structures [KS95, YW03] are introduced based on memory or disk. More details can be found in Section 6.3.2.

Although the previous approaches can be adopted by WCSs, new data structures and functionalities are needed. Moreover, as data in the WCS is normally distributed in cluster, maintaining a distributed index (e.g. B-tree in a distributed fashion) is time-consuming and non-trivial.

As described in Section 5.2, calculating temporal aggregation with ITA and CTA needs three steps. The first one is to compute the temporal aggregation groups based on the non-temporal grouping attributes and time instants. Then, the aggregation function is applied to each group. Finally, identical aggregation values with consecutive time points are coalesced together to generate so-called *constant interval* [JG09]. A constant interval is a maximal interval in which all values in that interval are identical. Instead of constructing the constant intervals after evaluating the aggregation function, we can also build the constant intervals after generating the aggregation groups. In consequence, each aggregation group is associated with a TI instead of a time instant. In some literatures [Tum92, JG09], such approach is called "Two-scans".

To incrementally recompute the temporal aggregation results, we can first incrementally update the aggregation groups, and then recompute the aggregation values based on modified groups. To achieve that, we utilize a data structure called "inverted index". The inverted index is mostly used for search engine indexing. It denotes which keywords contain in which files (web pages). Abstractly, an inverted index can be considered as a variant of hash-map in which the key is the content and the value is a list of its locations. In a WCS, an inverted index can be stored as a normal WCS table. Figure 8.5 shows its structure.

In Figure 8.5, the row key of the inverted index is composed by the non-

GP/TI	Value: Keys	Value: Changed

Figure 8.5: Structure of the inverted index

temporal grouped attributes (denoted as GP) and TI. Column "Value:Keys" maintains a list of $srk\#Value(A')$ where srk denotes row key of the base relation and $Value(A')$ indicates value of non-temporal attribute (A') (which the aggregation function is later applied to). "Value:Changed" is utilized to indicate whether the aggregation group is modified. For incrementally maintaining the aggregation values, we first select the aggregation groups which are tagged as "modified" and then update the aggregation values based on each changed group.

In the following sections, we will introduce how to use the "inverted index" to incrementally maintain the aggregation groups based on instantaneous temporal aggregation (ITA) and cumulative temporal aggregation (CTA). Temporal deltas are classified as deletions ∇ and insertions Δ.

Instantaneous temporal aggregation

Network

RK/TI	Network: Speed	Network: Supplier
Tom/5/∞	1000K	1&1

Inverted index

GP/TI	Value: Keys	Value: Changed
1&1/1/5		F
1&1/5/∞	Tom/5/∞#1000K	F

Δ

RK/TI	Network: Speed	Network: Supplier
Jim/4/7	900K	1&1

AVG

RK/TI	Speed: AVG
1&1/1/5	0
1&1/5/∞	1000K

Networkn

RK/TI	Network: Speed	Network: Supplier
Tom/5/∞	1000K	1&1
Jim/4/7	900K	1&1

Figure 8.6: Initial state of Network and inverted index

Suppose we have a table "Network" (shown in Figure 8.6) and want to calculate the daily average Internet speed of different Internet suppliers. Such query is represented as $AVG_{Speed}\gamma^{T}_{(Supplier,DAY,0)}$ $(Network)$ and its result is stored in "AVG" table.

Initially, there is only one tuple in "Network" table and the timeline of Internet supplier "1&1" is divided into two parts, namely, $[1, 5)$ and $[5, \infty)$. Correspondingly, the inverted index contains two entries, namely "1&1/1/5" and "1&1/5/∞". We set the contents of "Value:Changed" column to F to denote no change has been made. At a certain time, an insertion Δ is detected and "Network" is updated to "Network$_n$".

To update the inverted index, Algorithm 22 is performed. For each delta item t in Δ, when it contains the same non-temporal grouped attributes (denoted as A) with GP (line 7), three different situations can be distinguished.

1. $t.TI$ and $i.TI$ completely overlap.

2. $t.TI$ and $i.TI$ partially overlap.

3. $t.TI$ and $i.TI$ have no overlap.

For the first situation, we delete the contents of column "Value:Keys" and set the value of column "Value:Changed" to T (lines 8-10). For the second situation, it denotes the index entry i needs to be split. To split i, we distinguish whether ts or te is contained in $[is, ie)$ (lines 11-20). For the third situation, no changes are made to such group.

When t finds no index entries which share the same non-temporal grouped attributes (line 22), the whole timeline based on $t.A$ will be split (lines 23-26). Finally, for each index entry whose "Value:Changed" is T, contents of column "Value:Keys" will be updated (lines 30-34).

After updating the inverted index, the next step is to modify the temporal aggregation values. To achieve that, Algorithm 23 is used. We first choose the index entries which are tagged as changed (lines 1-2). For each changed index entry i, when a tuple t in the aggregation values has TI-overlapping with i and share the same non-temporal grouped attributes ($t.GP = i.GP$), t will be deleted and a new aggregation value based on i is produced (lines 4-8). Otherwise, new aggregation value is generated without discarding any old values (lines 10-12).

Figure 8.7 shows how to use the previous algorithms to incrementally maintain the temporal aggregation based on Figure 8.6. The numbers represent the order of the processing tasks.

Algorithm 24 describes how to maintain the inverted index based on deletions ∇. For each index entry i, when it has TI-overlapping with t and share the same value with $t.A$, value $srk\#Value(A')$ will be deleted from column

Algorithm 22 Maintaining inverted index (insertion)

```
1: for each tuple t in Δ do
2:     ts = t.TI.Start;
3:     te = t.TI.End;
4:     for each index entry i in Index do
5:         is = i.TI.Start;
6:         ie = i.TI.End;
7:         if i.GP = t.A then
8:             if ts ≦ is ∧ ie ≦ te then
9:                 Delete the contents of column "Value:Keys";
10:                Set column "Value:Changed" to T (true);
11:            else if is < ts < ie ∧ te > ie then
12:                Delete i;
13:                Insert a new tuple with row key i.GP/is/ts;
14:                Insert a new tuple with row key i.GP/ts/ie;
15:                Set column "Value:Changed" for both tuples to T;
16:            else if is < te < ie ∧ ts < is then
17:                Delete i;
18:                Insert a new tuple with row key i.GP/is/te;
19:                Insert a new tuple with row key i.GP/te/ie;
20:                Set column "Value:Changed" for both tuples to T;
21:            end if
22:        else
23:            Insert a new tuple with row key t.A/1/ts;
24:            Insert a new tuple with row key t.A/ts/te;
25:            Insert a new tuple with row key t.A/te/∞;
26:            Set column "Value:Changed" for these three tuples to T;
27:        end if
28:    end for
29: end for
30: for each index entry i in index do
31:    if i[Value : Changed] = T then
32:        Fetching srk#Value(A′) from updated base relations;
33:    end if
34: end for
```

Algorithm 23 Updating aggregation functions

```
1: for each index entry i in index do
2:     if i[Value : Changed] = T then
3:         for each t in aggregation values do
4:             if t.TI ∩ i.TI ≠ ∅ ∧ t.GP = i.GP then
5:                 Delete t;
6:                 Calculate aggregation values based on i;
7:                 Insert new generated aggregation value based on i;
8:                 Reset i[Value : Changed] to F (false);
9:             else
10:                Calculate aggregation values based on i;
11:                Insert new generated aggregation value based on i;
12:                Reset i[Value : Changed] to F (false);
13:            end if
14:        end for
15:    end if
16: end for
```

1. Split index entries

GP/TI	Value: Keys	Value: Changed
1&1/1/4		T
1&1/4/5		T
1&1/5/7		T
1&1/7/∞		T

2. Fetch srk#Value(A′) from Speed$_n$

GP/TI	Value: Keys	Value: Changed
1&1/1/4		T
1&1/4/5	Jim/4/7#900K	T
1&1/5/7	Jim/4/7#900K, Tom/5/∞#100K	T
1&1/7/∞	Tom/5/∞#1000K	T

3. Update aggregation values

GP/TI	Speed: AVG
1&1/1/4	0
1&1/4/5	900K
1&1/5/7	1450K
1&1/7/∞	1000K

4. Reset Value:Changed

GP/TI	Value: Keys	Value: Changed
1&1/1/4		F
1&1/4/5	Jim/4/7#900K	F
1&1/5/7	Jim/4/7#900K, Tom/5/∞#100K	F
1&1/7/∞	Tom/5/∞#1000K	F

Figure 8.7: Example of incremental recomputation of Δ

"Value:Keys" (lines 5-8). After index entries are updated, an "index-merge" operation is performed to combine the index entries which share the same non-temporal values (except the "Value:Changed" column) and their TIs are adjacent. At last, Algorithm 23 can be utilized to update the aggregation

Algorithm 24 Maintaining inverted index (deletion)

```
1: for each tuple t in ∇ do
2:     s = t.TI.Start;
3:     e = t.TI.End;
4:     for each index entry i in Index do
5:         is = i.TI.Start;
6:         ie = i.TI.End;
7:         if (ts ≦ ie ∧ is ≦ te) ∧ i.GP = t.A then
8:             Delete srk#Value(A′) in column "Value:Keys";
9:         end if
10:    end for
11: end for
12: Merging index entries and set column "Value:Changed" in new index
    entries to T;
```

values.

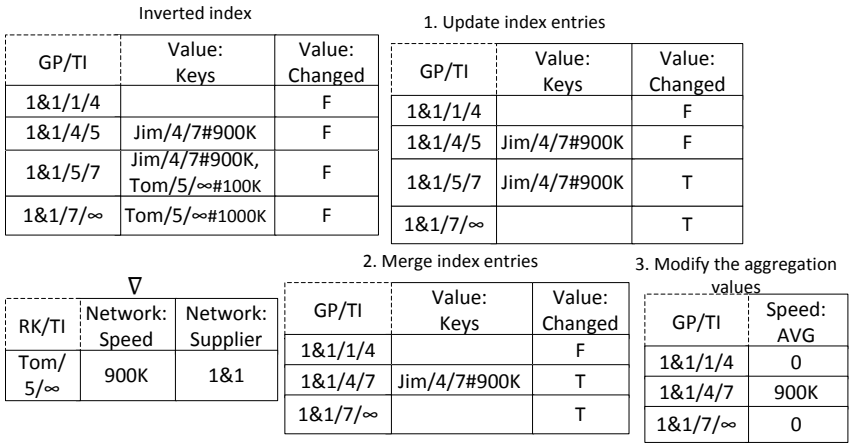


Figure 8.8: Example of incremental recomputation of ∇

Figure 8.8 shows an example of how to use Algorithm 24 incrementally recompute the temporal aggregation values. After updating the inverted index, two index entries "1&1/4/5" and "1&1/5/7" share the same non-temporal attribute values (except column "Value:Changed") and their TIs are adjacent. Consequently, these two index entries will be coalesced into one. For column "Value:Changed", its value in tuple "1&1/4/5" is F and in

tuple "1&1/5/7" is T. For coalesced tuple "1&1/4/7", we set its value to T (this is done by using logical operator *or*).

Cumulative temporal aggregation

To specify cumulative temporal aggregation (CTA), users need to set a moving window w. For each time instant t, its aggregation value is evaluated based on tuples which are valid during $[t-w, t+1)$ (To be consistent with our TI representation, we represent $[t-w, t]$ as $[t-w, t+1)$). In consequence, when a tuple t is inserted into or deleted from a temporal table, it will affect time points during $[t.TI.Start, t.TI.End+w)$. For example, when tuple "(Jim/4/7, 900K, 1&1)" is inserted into "Network" table (shown in Figure 8.6) and w is set to 2. Obviously, time points during $[4, 7)$ are affected. For time points 7 and 8, we should choose tuples which are valid during $[5, 8)$ and $[6, 9)$. As tuple "Jim/4/7" has TI-overlapping with them, it is also valid at time instants 7 and 8. Hence, it affects time points during $[4, 9)$.

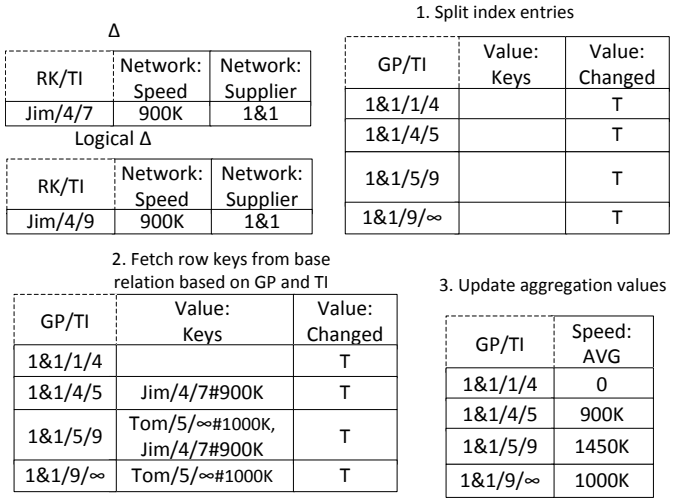


Figure 8.9: Example of incremental recomputation of CTA

Based on the previous example, incrementally maintaining CTA can be transformed to incrementally maintaining ITA. To achieve that, we transform the TI of each inserted or deleted tuple t into $[t.TI.Start,\ t.TI.End + w)$ where w denotes the moving window. Suppose we have the following query and Figure 8.9 shows the delta processing steps.

Query 3. *What is the average Internet speed of different Internet suppliers last two days?*

This query is specified as $AVG_{Speed}\gamma^{T}_{(Supplier,DAY,2)}\ (Network)$. The initial state of "Network" and query result can be found in Figure 8.6. For inserted tuple "(Jim/4/7, 900K, 1&1)", we transform its TI based on w, namely, $[4, 7 + 2)$. Logically, we can consider that a new tuple "(Jim/4/9, 900K, 1&1)" is inserted into "Network" table. When processing such insertion, Algorithms 22 and 23 can be reused.

8.2.5 Performance

The experiments are performed on a six-node cluster (Xeon QuadcoreCPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) running Hadoop (version 1.0.4) and HBase (version 0.94), respectively. Two temporal tables, namely, *Customers* (C) and *Orders* (O) (drawn in Figure 8.10) are generated based on a temporal data benchmark [KFea14] which is an extension of TPC-H. Initially, both tables contains 1 million rows.

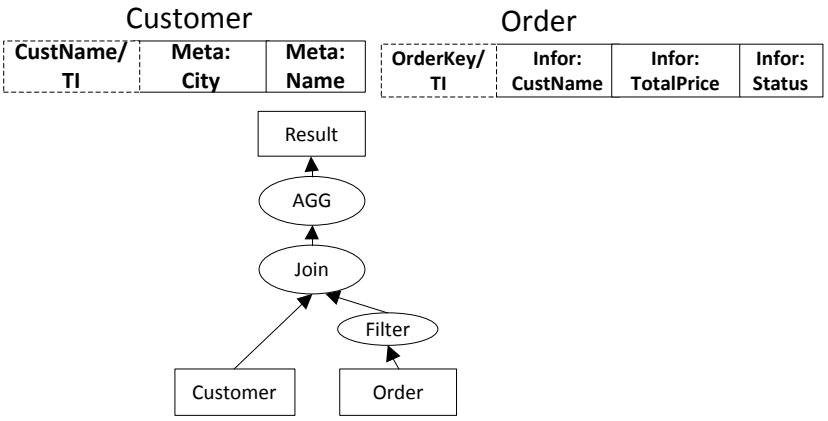


Figure 8.10: Schema definition and the abstract syntax tree

We evaluate the performance comparisons of full recomputation and incremental recomputation based on the following query.

Query 4. *Display the average daily consumption for various cities in which each individual order is paid and greater than 80 euro.* The corresponding TTRO representation is as follows:

1. $S = C \bowtie^{T}_{C.CustName=O.CustName} (\sigma^{T}_{(O.Status=1)\wedge(O.Price>80)}(O);$

2. $AVG_{price}\gamma^{T}_{(City,day,0)}(S)$.

As temporal aggregation is contained in the query, temporal deltas are classified as deletions ∇ and insertions Δ. Initially, the result of such query is calculated by three MapReduce jobs. The first one is to perform filter and join operation and the second and third ones are calculating the aggregation function.

To incrementally recompute the join operation, 6 MapRduce jobs need to be executed for calculating $\Delta_C \bowtie^T \Delta_N$, $\Delta_C \bowtie^T N_0$, $C_0 \bowtie^T \Delta_N$, $\nabla_C \bowtie^T \nabla_N$, $\nabla_C \bowtie^T N_0$ and $C_0 \bowtie^T \nabla_N$. For the temporal aggregation, two MapReduce jobs are implemented, namely, one for maintaining the inverted index and the other for updating the aggregation values. The performance comparisons of the incremental recomputation and the full recomputation is depicted in Figure 8.11.

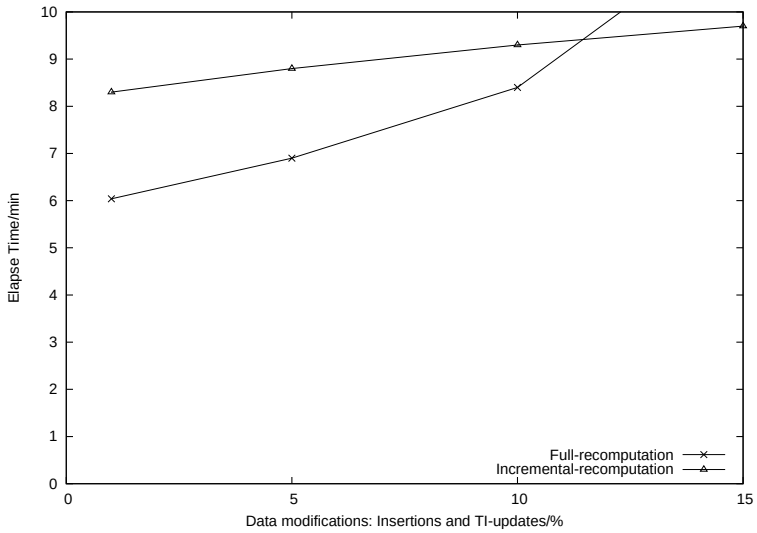


Figure 8.11: Performance between full recomputation and incremental recomputation

The incremental recomputation outperforms the full recomputation when the percentage of modified tuples is less than 12%. For the incremental recomputation, even with a small amount deltas (with 1%), its execution time is more than 6 minutes. The reason is that the incremental recomputation includes totally 9 MapReduce jobs, i.e. one for filter, six for join and two for aggregation. In our cluster, delivering MapReduce jar files into the

corresponding nodes needs 30-40 seconds. In consequence, only delivering MapReduce jar files in the cluster costs 4-5.4 minutes. Notice that, we use simple table-scan to implement the incremental and full recomputations. The main purpose of this performance comparison is to show the correctness of our algorithms and heuristic query optimization, e.g. predicate push-down, is performed.

8.3 Incremental temporal recomputation with CTO operators

In the previous section, we introduced the incremental temporal recomputation with $TTRO$ operators. To make delta propagation closed, temporal queries are classified into "snapshot-reducible" and "extended snapshot-reducible", respectively. For the snapshot-reducible queries, TI-updates □ can be propagated as update. For the extended-snapshot reducible queries, □ needs to be treated as deletion/insertion pairs.

In this section, we look at how to incrementally recalculate results evaluated by CTO operators. The CTO operator model is proposed to process the *explicit history representation* in which each data version is associated with an explicit temporal interval (TI) (attribute-timestamping model). To incrementally maintain temporal query results based on the CTO operator model, three approaches are possible:

1. Transform the *explicit history representation* and CTO operators into their non-temporal counterparts and reuse non-temporal view maintenance approaches. To achieve that, we can utilize the *tuple-timestamping representation* and $TTRO$ operators as the intermediate layer.

2. Transform the *explicit history representation* and CTO operators into *tuple-timestamping representation* and $TTRO$ operators and reuse the approaches described in Section 8.2.

3. Directly propagate attribute-timestamping deltas without additional temporal data (including both base tables and data changes) and temporal query transformations.

Clearly, the first and second approaches can lead to expensive temporal data transformations and complicated temporal query translations. Hence, we prefer the third one.

For incremental recomputation based on the CTO operators, *enhanced attribute-level delta representation* is used. In a highly abstract level, the enhanced attribute-level delta representation can be considered as an alternative of the row-level delta representation. Hence, it is possible to reuse some of the temporal delta propagation rules defined for tuple-timestamping representation. However, several new challenges and optimization opportunities arise.

- As enhanced attribute-level representation maintains meta-data for each changed column, we can utilize this information to optimize the delta propagation. For example, when a new generated data version is not included in the projection attribute, there is no need to perform the incremental view maintenance procedure over that data.

- Based on the state of materialized view and temporal query predicates, a version modification can cause either a version modification or modifications for multi-versions, e.g. a version deletion can lead to a tuple deletion. In consequence, delta propagation procedures should distinguish those different types of outputs.

- As temporal delta is represented at the attribute-level, a mixed output will be generated for join operator, e.g. a combination of a version-insertion and a version-TI-update.

The following sections address the issues mentioned above and describe the algorithms for incrementally maintaining the temporal views based on the snapshot-reducible and extended snapshot-reducible queries, respectively.

8.3.1 Snapshot-reducible queries

The main characteristics of a snapshot-reducible query is that it does not explicitly reference the time in the predicate. In consequence, $\square$ can be propagated as update itself.

Project π^c_A

To incrementally recompute π^c_A operator, we introduce a new function $Check$. The functionality of $Check$ is to test whether the modified column is referenced in projection attributes A. When $Check$ returns true, Δ and $\square$ can be propagated as $\pi^c_A(\Delta)$ and $\pi^c_A(\square)$, respectively. Otherwise, no output will be generated. Table 8.7 shows the incremental procedures for π^c_A.

Delta_Type	Check	Modifications
Δ	F	
Δ	T	$\pi^c_A(\Delta)$
$\square$	T	$\pi^c_A(\square)$
$\square$	F	

Table 8.7: Procedures for maintaining π^c_A

Filter σ^c_p

Different from π^c_A, we cannot simply discard the delta items when they are not referenced in the filter predicate p. The reason is that it is possible that its correlated unchanged columns are satisfied with p. When $Check(\Delta)$ and $Check(\square)$ are false, such delta item can only cause single version-insertion or single version-TI-update.

However, when $Check(\Delta)$ and $Check(\square)$ are true, a version-insertion and a version-TI-update can cause insertions and TI-updates for multi-versions. In consequence, to distinguish those different operations, we use I_{mv} and μ_{mv} to denote view modifications for multiple versions and I_v and μ_v for single version. I and μ represent insertion and TI-update, respectively. The incremental procedures of σ^c_p is described in Table 8.8.

After executing the incremental temporal delta processing, coalesce operation $\boxplus$ (defined in Section 5) should be performed to combined data versions in each column which are value-equivalent and their TIs are adjacent.

Delta_Type	Check	Modifications
Δ	T	$I_{mv}(\sigma^c_p(\Delta))$
Δ	F	$I_v(\sigma^c_p(\Delta))$
$\square$	T	$\mu_{mv}(\sigma^c_p(\square))$
$\square$	F	$\mu_v(\sigma^c_p(\square))$

Table 8.8: Procedures for maintaining σ^c_p

Figure 8.12 shows this example. Suppose we select Internet suppliers whose speed is greater than or equal to 1000K. Q_0 denotes the initial state. At time point 10, "1000K" is updated into "1100K". $Check(\Delta_n) = T$ and $Check(\square_n) = T$. In consequence, Δ_{mv} and $\square_{mv}$ are generated. Notice that, for better understanding, Δ_{mv} and $\square_{mv}$ are represented at row-level. Q_{ou} denotes the state of Q_o after performing the updates. For column "Network:Supplier" in Q_{ou}, versions "1&1:[5,10)" and "1&1:[10,∞)"

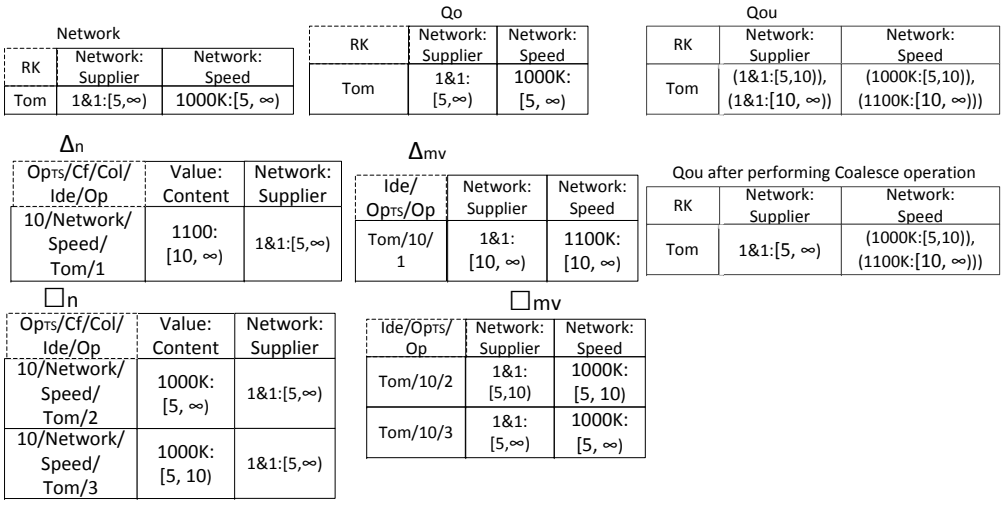


Figure 8.12: Performing ⊞ after executing incremental procedures

are value-equivalent and their TIs are adjacent. Hence, they are combined to "1&1:[5,∞)".

Join $\bowtie_p^c$

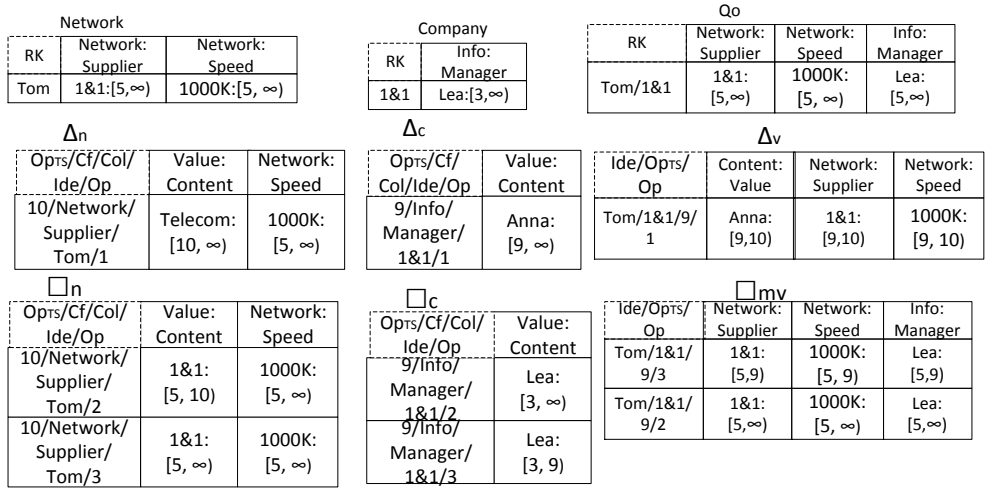


Figure 8.13: Example for incremental join procedures

To incrementally maintain the temporal view $R \bowtie_p^c S$, two join tables are

decomposed as R_0, Δ_R, $\square_R$, S_0, Δ_S and $\square_S$ where R_0 and S_0 represent the unchanged data from R and S, respectively. The incremental procedure for $R \bowtie_p^c S$ is described in Table 8.9. For better readability, we use $Schema(\bowtie_p^c)$ to denote the schema of $R \bowtie_p^c S$ and CC is a shorthand to reference the name of the changed column.

In Table 8.9, three different situations are distinguished:

1. In both operands, join columns were modified.
2. Only in one operand, a join column was modified.
3. No join columns were modified.

R	**S**	**Check**$_R$	**Check**$_S$	**Modifications**
R_0	S_0			
R_0	Δ_S	T	T	$I_{mv}(R_0 \bowtie_p^c \Delta_S)$
R_0	Δ_S	T	F	$I_v(R_0 \bowtie_p^c \Delta_S)$
R_0	$\square_S$	T	T	$\mu_{mv}(R_0 \bowtie_p^c \square_S)$
R_0	$\square_S$	T	F	$\mu_v(R_0 \bowtie_p^c \square_S)$
Δ_R	Δ_S	T	T	$I_{mv}(\Delta_R \bowtie_p^c \Delta_S)$
Δ_R	Δ_S	T	F	$I_{mv}(\Delta_R \bowtie_p^c \Delta_S)$
Δ_R	Δ_S	F	T	$I_{mv}(\Delta_R \bowtie_p^c \Delta_S)$
Δ_R	Δ_S	F	F	$I_{mv}(\pi^c_{\Delta_R.CC,\Delta_S.CC}(\Delta_R \bowtie_p^c \Delta_S))$
Δ_R	$\square_S$	T	T	$I_{mv}(\Delta_R \bowtie_p^c \square_S^+)$
Δ_R	$\square_S$	T	F	$I_{mv}(\Delta_R \bowtie_p^c \square_S^+)$
Δ_R	$\square_S$	F	T	$\mu_{mv}(\pi^c_{Schema(\bowtie_p^c)-\Delta_R.CC}(\Delta_R \bowtie_p^c \square_S)),$ $I_v(\pi^c_{\Delta_R.CC}(\Delta_R \bowtie_p^c \square_S^+))$
Δ_R	$\square_S$	F	F	$\mu_v(\pi^c_{\square_R.CC}(\Delta_R \bowtie_p^c \square_S)),$ $I_v(\pi^c_{\Delta_S.CC}(\Delta_R \bowtie_p^c \square_S^+))$
$\square_R$	$\square_S$	T	T	$\mu_{mv}(\square_R \bowtie_p^c \square_S)$
$\square_R$	$\square_S$	T	F	$\mu_{mv}(\square_R \bowtie_p^c \square_S)$
$\square_R$	$\square_S$	F	T	$\mu_{mv}(\square_R \bowtie_p^c \square_S)$
$\square_R$	$\square_S$	F	F	$\mu_{mv}(\pi^c_{\square_R.CC,\square_S.CC}(\square_R \bowtie_p^c \square_S))$

Table 8.9: Procedures for maintaining $\bowtie_p^c$

We use an example in Figure 8.13 to illustrate how the procedures in Table 8.9 can be used. Suppose we have $N \bowtie^c_{N.Supplier=C.RK} C$, where N and C denote "Network" and "Company", respectively. Q_o represents the initial join

result. For table "Network", $Check(\Delta_n) = T$ and $Check(\square_n) = T$. As row key of "Company" is used in join predicate, every version-modification will implicitly modify the TI of the row key, $Check(\Delta_c) = T$ and $Check(\square_c) = T$.

To calculate the inserted and TI-updated values for Q_o,

- $\Delta_n \bowtie^c \Delta_c = \emptyset$.
- $\Delta_n \bowtie^c \square_c^+ = \emptyset$.
- $\square_n^+ \bowtie^c \Delta_c = \Delta_v$. Multi-version insertions "Anna:[9,10)", "1&1:[9,10)" and "1000K:[9,10)" are generated.
- $\square_n \bowtie^c \square_c = \square_{mv}$. TIs of all data versions belong to tuple "Tom/1&1" will be modified from $[5, \infty)$ to $[5, 9)$.

As same discussions made for σ_p^c, coalesce operation $\boxplus$ will be performed after executing the incremental temporal delta processing.

8.3.2 Extended Snapshot-reducible queries

When temporal query q is extended snapshot-reducible, q has an explicit reference to time and the temporal delta propagations are not closed. To overcome such issue, we represent the extracted deltas as deletions ∇ and insertions Δ.

Project π_A^c

The incremental procedures for π_A^c operator is described in Table 8.10.

Delta_Type	Check	Modifications
Δ	F	
Δ	T	$\pi_A^c(\Delta)$
∇	T	$\pi_A^c(\nabla)$
∇	F	

Table 8.10: Procedures for maintaining π_A^c

Filter σ_p^c

The incremental procedures for σ_p^c operator is described in Table 8.11. As same as the version-insertion, a version deletion can also cause deletions for multi-versions. In consequence, we use D_{mv} and D_v to represent multi-version deletions and single version deletion, respectively.

Delta_Type	**Check**	**Modifications**
Δ	T	$I_{mv}(\sigma_p^c(\Delta))$
Δ	F	$I_v(\sigma_p^c(\Delta))$
∇	T	$D_{mv}(\sigma_p^c(\nabla))$
∇	F	$D_v(\sigma_p^c(\nabla))$

Table 8.11: Procedures for maintaining σ_p^c

Join $\bowtie_p^c$

To incrementally maintain the temporal join views $R \bowtie_p^c S$, the join operands are decomposed as R_0, Δ_R, ∇_R, S_0, Δ_S and ∇_S. The incremental procedure for $R \bowtie_p^c S$ is described in Table 8.12.

R	**S**	**Check**$_R$	**Check**$_S$	**Modifications**
R_0	∇_S	T	T	$D_{mv}(R_0 \bowtie_p^c \nabla_S)$
R_0	∇_S	T	F	$D_v(\pi^c_{\nabla_S.CC}(R_0 \bowtie_p^c \nabla_S))$
Δ_R	∇_S	T	T	
Δ_R	∇_S	T	F	$I_{mv}(\pi^c_{schema(\bowtie_p^c)-\nabla_S.CC}(\Delta_R \bowtie_p^c \nabla_S))$
Δ_R	∇_S	F	T	$D_{mv}(\pi^c_{schema(\bowtie_p^c)-\Delta_R.CC}(\Delta_R \bowtie_p^c \nabla_S))$
Δ_R	∇_S	F	F	$I_v(\pi^c_{\Delta_R.CC}(\Delta_R \bowtie_p^c \nabla_S))$, $D_v(\pi^c_{\nabla_S.CC}(\Delta_R \bowtie_p^c \nabla_S))$
∇_R	∇_S	T	T	$D_{mv}(\nabla_R \bowtie_p^c \nabla_S)$
∇_R	∇_S	T	F	$D_{mv}(\nabla_R \bowtie_p^c \nabla_S)$
∇_R	∇_S	F	T	$D_{mv}(\nabla_R \bowtie_p^c \nabla_S)$
∇_R	∇_S	F	F	$D_v(\pi^c_{\nabla_R.CC}(\nabla_R \bowtie_p^c \nabla_S))$, $D_v(\pi^c_{\nabla_S.CC}(\nabla_R \bowtie_p^c \nabla_S))$

Table 8.12: Procedures for maintaining $\bowtie_p^c$

The temporal delta propagation procedures for $R_0 \ltimes_p^c \Delta_S$, $S_0 \ltimes_p^c \Delta_R$ and $\Delta_R \ltimes_p^c \Delta_S$ can be found in Table 8.9. After the incremental delta processing is performed, coalesce operation $\boxplus$ will be executed.

8.3.3 Example

In this section, we give an example to combine various incremental procedures together. Suppose we have two tables "Network" (N) and "Company" (C) (See Figure 8.14) and the following temporal view definition:
$V = \sigma^c_{N.Speed<1000 \wedge duration(N.Speed.TI)<10}(N) \ltimes^c_{N.Supplier=C.RK} C$.

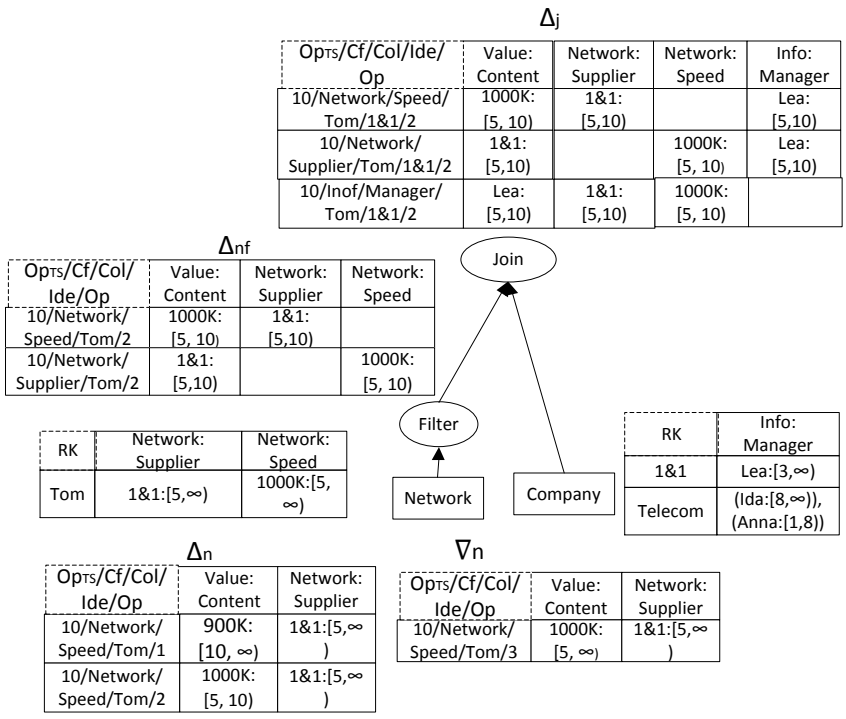


Figure 8.14: Example of temporal view maintenance with ATM

The abstract syntax tree and the corresponding data (including base relations and temporal deltas) are shown in Figure 8.14. As the view definition contains the temporal comparisons ($duration(TI)$), it is classified to the extended snapshot-reducible. Hence, the temporal deltas are represented as deletion ∇ and insertion Δ. Initially, V is empty. At time

point 10, 1000K is updated to 900K. To propagate such delta, $Check(\Delta_n) = true$ and $Check(\nabla_n) = true$. For the filter operation, delta item "10/Network/Speed/Tom/2" satisfies the predicate and multi-version insertions are produced (the results are denoted as Δ_{nf}). For the temporal join operation, $Check(\Delta_{nf}) = true$. Table "Company" is viewed as unchanged data, i.e. C_0. We calculate $\Delta_{nf} \bowtie^c C_0$ and Δ_j is generated.

8.3.4 Temporal aggregation $f_{A'}(\gamma^c_{(A,TG,W)})$

To incrementally recompute temporal aggregation $f_{A'}(\gamma^c_{(A,TG,W)})$, inverted index described in Section 8.2.4 can be reused.

Figure 8.15 shows an example based on an instantaneous temporal aggregation $AVG_{Speed}(\gamma^c_{(Supplier,day,0)})$. For every index entry i which is tagged as "T", we first locate tuple "1&1" in "AVG" (using $i.GP$). Data value "(1000K:[5,∞)" in column "Speed:AVG" is deleted as it has TI overlapping with "1&1/5/10" in inverted index. At last, new aggregation values based on index entries "1&1/5/10" and "1&1/10/∞" are generated.

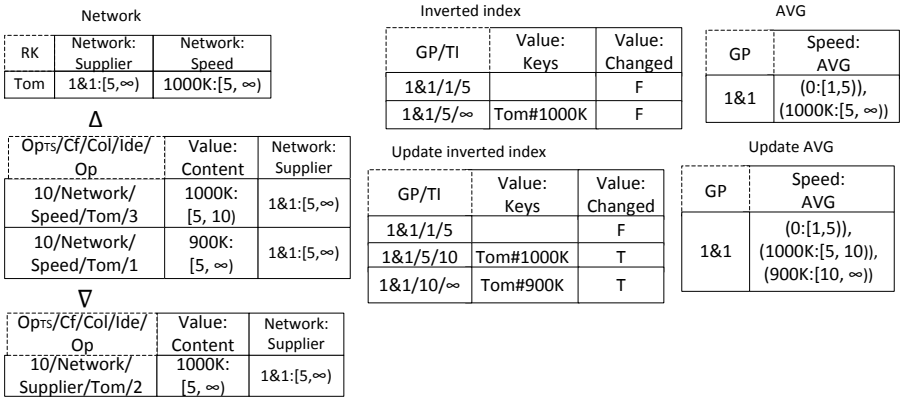


Figure 8.15: Incremental recomputation for $AVG_{Speed}(\gamma^c_{(Supplier,day,0)})$

Similar to Project π^c_A, when the modified columns of delta items are not referenced in A or A', those data changes can be ignored.

8.3.5 Performance

We utilize the same hardware configurations and temporal query described in Section 8.2.5 to evaluate the performance of the incremental recomputation

with CTO operators. The enhanced attribute-level-representation is used to represent each delta item.

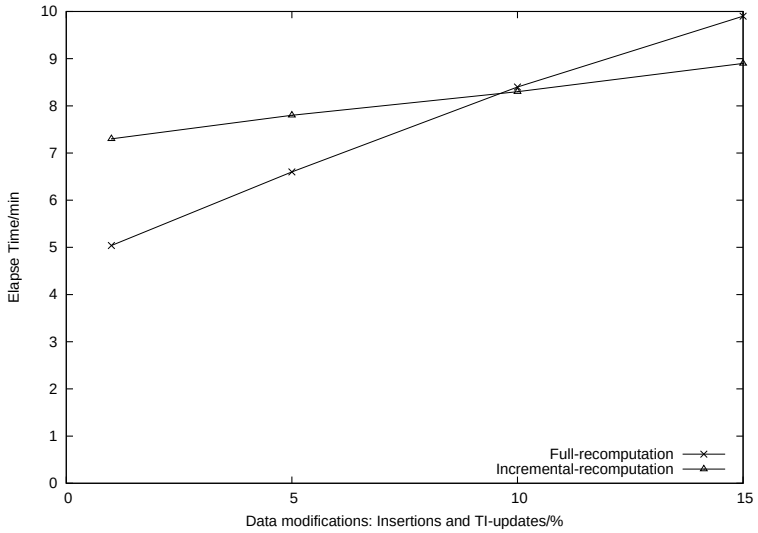


Figure 8.16: Performance comparisons between the incremental reocmputation and the full recomputation based on CTO operators

Figure 8.16 shows the performance comparisons between the incremental reocmputation and the full recomputation. The incremental recomputation outperforms the full recomputation when the portion of change data is less than 10%.

8.4 Summary

In this chapter, we introduced the temporal delta propagation rules based on $TTRO$ and CTO operators. Instead of representing the complete temporal deltas as deletions ∇ and insertions Δ, we represent the complete deltas as insertions Δ and TI updates $\square$. As a WCS table usually maintains a large number of columns, treating the update as an individual class can be more efficient when only a small number of columns are modified/updated.

However, as already shown in Section 8.1, propagating $\square$ can cause deletions which lead to the delta propagation not *closed*, namely, a new delta type is generated through delta propagation. To solve such problem, we

classified temporal queries as "snapshot-reducible" and "extended snapshot-reducible". To maintain the snapshot-reducible queries, an update can be propagated as update itself. For the extended snapshot-reducible queries, an update needs to be propagated as a deletion/insertion pair.

For propagating temporal deltas, row-level-representation and enhanced attribute-level-representation are used for $TTRO$ and CTO operator models, respectively. At a highly abstract level, both represent deltas in a row-oriented way. In consequence, change data propagation rules proposed for the $TTRO$ operators are also feasible for the CTO operators.

However, for the CTO operators, as enhanced attribute-level-representation contains the name of the modified column in row key, we can optimize the incremental algorithms by first checking whether the modified columns are referenced in the temporal query. Moreover, as a version-modification can cause either a version-modification or modifications of multiple versions, we classified the types of temporal view modification as single version-insertion I_v, single version-update μ_v, single version-deletion D_v, multi-version insertions I_{mv}, multi-version updates μ_{mv} and multi-version deletions D_{mv}. After executing the incremental delta processing, coalesce operation $\boxplus$ is performed to combine data versions which are value-equivalent and TIs are adjacent.

To incrementally maintain the temporal aggregation based on instantaneous temporal aggregation and cumulative temporal aggregation, we explored a data structure called *inverted index*. Each inverted index can be stored as a normal WCS table and no additional modifications are needed. Comparing to the existed approaches, maintaining inverted index is straightforward and easier.

The performance charts (based on $TTRO$ and CTO operator models) have shown that the incremental recomputation outperforms the full recomputation when the portion of change data is less than 15%.

Chapter 9

Incremental temporal data recomputation with partial temporal deltas

In general, change-data capture (CDC) approaches cannot always generate complete delta sets because of their limitations, the available information provided by the data sources and the performance requirements (see descriptions in Chapter 7).

In the previous chapter, we introduced how to incrementally recompute materialized temporal query results based on complete temporal deltas. In this chapter, we will address these issues for (partially) incomplete deltas. We first introduce three general methods for propagating partial temporal deltas and discuss their limitations and drawbacks. Then, we explain how to propagate partial temporal deltas based on the $TTRO$ and CTO operator models, respectively.

9.1 Propagating methods

Traditional incremental recomputation approaches for both non-temporal and temporal deltas require complete delta sets. However, as we have already seen in Chapter 7, complete delta sets cannot always be generated. Generally, propagating partial (incomplete) deltas is only possible when certain information is provided. For example, suppose we maintain a materialized view $V = \pi_A(R)$ where π is the non-temporal Project operator. Our CDC

approach detects a partial update delta item t_{pup}. t_{pup} can only be propagated if the row key is included in the projected results. In commercial database systems, e.g. MySQL, such functionality can be performed via "*INSERT ... ON DUPLICATE KEY UPDATE*" syntax. However, when the row key is updated, previous partial delta processing is impossible.

In the non-temporal context, authors in [JD11] studied how to maintain the state of a data warehouse (DW) based on the partial non-temporal deltas. They proposed an incremental maintenance approach with partial deltas for a specific type of view called *Dimension View*. A dimension view is a *SPJ* (Select-Project-Join) view in which each tuple in the view contains a unique identifier, i.e. a primary key (or composite primary key) and the join predicates should at least reference the primary keys of the base tables.

To propagate partial deltas into the dimension views, [JD11] first defines a set of transformation rules to transform partial deltas into complete deltas. After delta transformation is done, the existing view maintenance approaches are reused. Finally, the results of change data propagation will be converted back into the partial format.

When propagating partial deltas in the temporal context, three alternative approaches can be followed:

1. Transform temporal data (including partial temporal deltas and the corresponding base tables) and temporal queries into their non-temporal counterparts and reuse the approaches proposed by [JD11].

2. Transform the partial temporal deltas into complete temporal deltas by utilizing the temporal base relations stored in the data warehouse or external look-up tables, and then reuse the incremental approaches described in Chapter 8.

3. Propagate partial temporal deltas directly without additional temporal delta and temporal query transformations.

Although the third approach can be utilized in non-temporal databases, it turns out to be unsuitable for temporal data.

An example is shown in Figure 9.1. Suppose each column in "Network" table (at Source side) can only contain 2 data versions and we utilize the Audit-column approach (Suppose the insertion time for row "Tom" is 2) to extract deltas. The view definition is given as *Fast-network* = $\sigma^{T}_{Network:Speed \geq 1000}(Network)$, i.e. select Network suppliers whose speed is

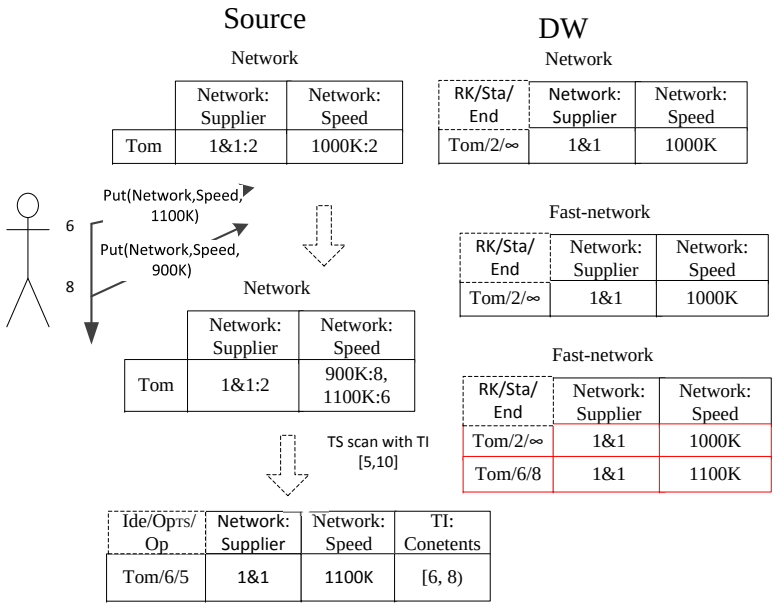


Figure 9.1: A counterexample for directly propagating partial temporal deltas

greater than or equal to 1000K. As "1&1" is the oldest data version in tuple "Tom", "1100K:6" is treated as a partial update (denoted as tuple "Tom/6/5"). To process delta item "Tom/6/5", it satisfies filter predicate and shares no row key with tuples in "Fast-network" table. Consequently, tuple "Tom/6/8" is inserted into the view. However, after insertion, tuple "Tom/2/∞" and tuple "Tom/6/8" have temporal conflicts.

As shown in the previous example, directly propagating partial temporal deltas without temporal delta transformation can lead to the wrong state of the result table. Hence, to incrementally propagate temporal partial deltas, we adopt the second of the above described approach.

Recall the definition of partial temporal delta set in Chapter 7. An inserted tuple which is detected at the data source will lead to an insertion for its corresponding base relation in data warehouse. An update and a deletion made at the data source will cause an insertion with a TI-update. *Partial update* means that CDC can only obtain the current state of the updated tuple whereas its previous state is unknown. Hence, we can transform the partial-update deltas as insert deltas in which the corresponding TI-update

deltas are unknown (or partially known). For transforming the upsert deltas, the same method as for partial-updates are used.

Based on the previous description, partial-update delta "6/Tom/5" in Figure 9.1 can be represented as insertion Δ and partial TI-update $\square^p$ in Figure 9.2.

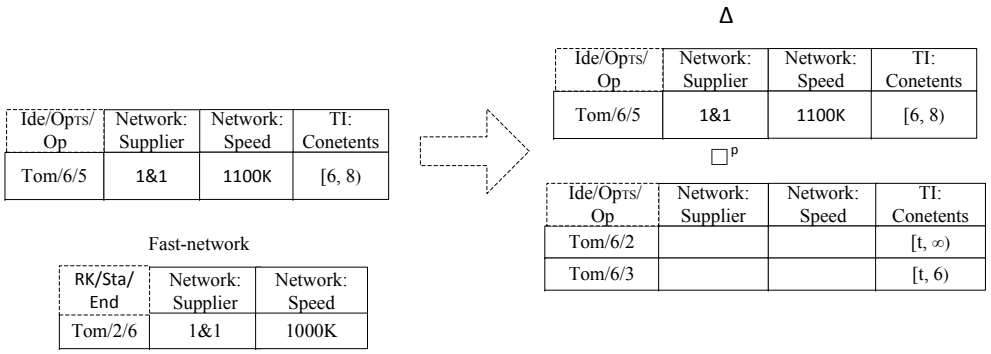


Figure 9.2: Partial temporal deltas transformation

In Figure 9.2, $\square^p$ is represented as a pair $\square^{p-}/\square^{p+}$. Values of column "Network:Speed" are empty and the starting-point of TI is tagged as unknown t. Clearly, both tuples in $\square^p$ cannot be checked by the filter predicate as values of "Network:Speed" are missing. However, we can use time point 6 with identifier "Tom" to update "Fast-network" table. As we find tuple "Tom/2/∞" in "Fast-network" which contains "Tom" in its row keys and time 6 is included in its TI, $TI.End$ of "Tom/2/∞" is updated to 6.

However, when temporal query contains temporal predicates, the previous strategy cannot be used, as the starting-point of TI is unknown. In consequence, partial TI-updates $\square^p$ need to be transformed into complete TI-updates $\square$ using contents of base relations in the data warehouse (DW).

Lemma 9.1.1. *Partial TI-updates $\square^p$ can be propagated by using the contents of materialized temporal views when temporal queries are snapshot-reducible. Otherwise, contents of base relations in the DW are needed for partial delta transformation.*

Proof. When q is snapshot-reducible, only non-temporal comparisons exist in q. Hence, $\square^p$ will only affect tuple t in materialized view when $t.TI$ contains $\square^{p+}.TI.End$ and share the same srk (row key from the source table). When such tuple t exists, $t.TI.End$ is modified to $\square^{p+}.TI.End$. As TIs in $\square^p$ are

unknown, when q is extended snapshot-reducible, it is impossible to evaluate temporal predicates in q. Hence, $\square^p$ needs to be translated into $\square$. $\square$

In the following sections, we describe how partial TI-update $\square^p$ can be propagated based on TTRO and CTO operators. Temporal delta propagation rules for insertions Δ can be found in Chapter 8.

9.2 Incremental temporal data recomputation with TTRO operators

Based on lemma 9.1.1, temporal queries are classified as snapshot reducible and extended snapshot-reducible.

9.2.1 Snapshot-reducible queries

For the snapshot-reducible queries, $\square^p$ can be propagated with the contents of materialized view.

Project π_A^T and Filter σ_p^T

Algorithm 25 Processing $\square^p$ with π_A^T and σ_A^T

```
1: for each tuple s in □^{p+} do
2:     for each tuple t in view do
3:         if s.srk = t.srk ∧ t.TI.Start ≦ s.TI.End < t.TI.End then
4:             t.TI.End = s.TI.End;
5:         end if
6:     end for
7: end for
```

Algorithm 25 describes how $\square^p$ can be processed with π_A^T and σ_A^T. For each tuple s in the after-state $\square^{p+}$, we utilize $s.srk$ and $s.TI.End$ as a selection criteria to search tuples in the materialized views. When such tuple t is found, $t.TI.End$ is updated to $s.TI.End$. Corresponding example can be found in Figure 9.2.

Notice that, for σ_p^T, as delta values are missing (see Figure 9.2), Algorithm 25 is not *net-effect*, namely, it is possible that tuple s in $\square^{p+}$ finds no corresponding tuple in the materialized view.

Join $\bowtie_p^T$

Algorithm 26 Processing $\square^p$ with $\bowtie_p^T$

```
1: for each tuple s in □^{p+} do
2:     for each tuple t in view do
3:         if t.srk.contains(s.srk) ∧ t.TI.Start ≦ s.TI.End < t.TI.End
   then
4:             t.TI.End = s.TI.End;
5:         end if
6:     end for
7: end for
```

Algorithm 26 describes how $\square^p$ can be processed based on $Join_p^T$. As $\bowtie_p^T$ has two operands, Algorithm 26 will be performed twice.

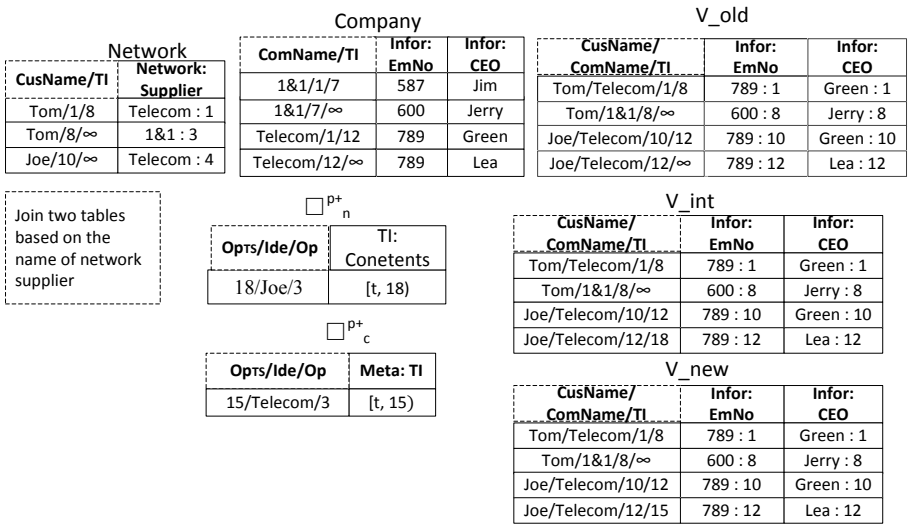


Figure 9.3: Processing $\square^p$ with $\bowtie_p^T$

We use an example in Figure 9.3 to illustrate Algorithm 26. Suppose we join tables "Network" and "Company" based on the name of network supplier. "V_old" denotes the initial state of the join result. To propagate partial TI-updates, we first run Algorithm 26 against $\square_n^{p+}$ (for "Network") (intermediate results stored in "V_int"), and then against $\square_c^{p+}$ (for "Company").

For delta item "18/Joe/3", TI of tuple "Joe/Telecom/12/∞" is updated to $[12, 18)$. For delta item "15/Telecom/3", TI of tuple "Joe/Telecom/12/18" (in table "V_int") is changed to $[12, 15)$.

9.2.2 Extended snapshot-reducible queries

For the extended snapshot-reducible queries, $\square^p$ needs to be transformed to $\square$ using the contents of base relations in DW. Algorithm 27 describes how $\square^p$ can be transformed to $\square$. (p_1, p_2) represents the before- and after-state of partial TI-update. For each tuple t in base relation, we detect whether t shares the same row key with p_2 and $p_2.TI.End$ is contained in $t.TI$ (line 3). When t is found, $p_1 = t$. For p_2, we assign the values of non-temporal attributes of t to p_2 and change $p_2.TI.Start$ to $t.TI.Start$ (line 5).

Algorithm 27 Transforming $\square^p$ to $\square$

```
1: for each pair (p1, p2) in □^p do
2:     for each tuple t in the corresponding base relation in DW do
3:         if p2.srk = t.srk ∧ t.TI.Start ≦ p2.TI.End < t.TI.End then
4:             p1 = t;
5:             p2.value = t.value and p2.TI = [t.TI.Start, p2.TI.End);
6:         end if
7:     end for
8: end for
```

For example, suppose we transform the partial-update delta item ("6/Tom /2", "6/Tom/3") in Figure 9.2. In "Network" table (on DW side), tuple "Tom/2/ ∞" shares the same srk "Tom" with "6/Tom/3" and its TI contains time 6. Hence, the before-state $\square^-$ is (Tom/2/∞, 1&1, 1000K) and the after-state $\square^+$ is (Tom/2/6, 1&1, 1000K). After partial temporal deltas are transformed into the complete temporal deltas, algorithms described in Chapter 8 can be used.

9.3 Incremental temporal data recomputation with CTO operators

When incrementally processing partial temporal deltas with CTO operators, temporal queries are classified as snapshot-reducible and extended snapshot-

reducible. As each delta item is represented at the level of attribute, we use function $Check$ to detect whether the modified column (denoted as CC in the following sections) is referenced in projection attributes or predicates.

9.3.1 Snapshot-reducible queries

Project π^c_A

Algorithm 28 Processing $\square^p$ with π^c_A

```
1: for each tuple s in □^{p+} do
2:     if Check(s) = true then
3:         for each tuple t in view do
4:             if t.srk = s.srk then
5:                 for each data version d in t[CC] do
6:                     if d.TI.Start ≦ s.TI.End < d.TI.End then
7:                         d.TI.End = s.TI.End;
8:                     end if
9:                 end for
10:            end if
11:        end for
12:    end if
13: end for
```

For each delta item s in $\square^{p+}$, when $Check(s)$ is true (line 2), we first locate tuple t which share the same srk with s (lines 3-4). Then, for each data version d in $t[CC]$, if $d.TI$ contains $s.TI.End$, $d.TI.End$ is updated to $s.TI.End$ (lines 5-9).

Filter σ^c_p and Join $\bowtie^c_p$

As same discussions made in Section 8.3, even if CC of a delta item is not referenced in p, the value of the materialized views can still be modified. Moreover, a version-modification can cause modifications of multiple data versions.

In Algorithm 29, we distinguish if CC in s is referenced in predicate p (lines 2-11) or not (lines 12-22). When CC is referenced in p, it denotes that data versions in t which contains $s.TI.End$ should be modified. When CC is not referenced in p, only single data version will be changed.

Algorithm 29 Processing $\square^{p}$ with σ_{p}^{c} and $\ltimes_{p}^{c}$

```
1: for each tuple s in □^{p+} do
2:     if Check(s) = true then
3:         for each tuple t in view do
4:             if t.srk.contains(s.srk) then
5:                 for each data version d in t do
6:                     if d.TI.Start ≦ s.TI.End < d.TI.End then
7:                         d.TI.End = s.TI.End;
8:                     end if
9:                 end for
10:            end if
11:        end for
12:    else
13:        for each tuple t in view do
14:            if t.srk.contains(s.srk) then
15:                for each data version d in t[CC] do
16:                    if d.TI.Start ≦ s.TI.End < d.TI.End then
17:                        d.TI.End = s.TI.End;
18:                    end if
19:                end for
20:            end if
21:        end for
22:    end if
23: end for
```

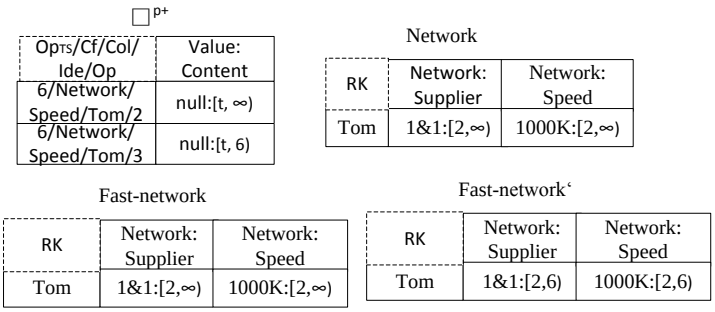


Figure 9.4: Processing $\square^{p}$ with σ_{p}^{c}

Figure 9.4 shows an example about how to propagate partial TI-update $\square^p$ with filter operator. Suppose we want to select the network suppliers whose speed is greater than or equal to 1000K , i.e. $\sigma^c_{Network:Speed \geqq 1000}(Network)$. Table "Fast-network" denotes the initial state of such query. To propagate $\square^p$, CC in delta item "6/Network/Speed/Tom/3" is referenced in predicate. Hence, all TIs in tuple "Tom" will be updated.

9.3.2 Extended snapshot-reducible queries

As same discussions made in Section 9.2.2, $\square^p$ has to be transformed into the complete TI-update deltas $\square$ for the extended snapshot-reducible queries. To achieve that, Algorithm 27 can be performed based on each column. After partial temporal deltas are transformed into the complete temporal deltas, algorithms described in Section 8 can be used.

9.4 Summary

In this chapter, we studied the problem of maintaining materialized temporal results with partial temporal deltas based on $TTRO$ and CTO operators, respectively. For propagating partial deltas, three approaches can be adopted.

However, according to our example (see the example in Section 9.1), directly propagating partial temporal deltas can lead to a wrong state of the result table. In consequence, partial temporal deltas need to be transformed into the complete temporal deltas. During delta transformation, upsert deltas and partial update deltas will be treated as insertions Δ where the values of TI-updates $\square$ are partially available. To propagate partial TI-update deltas $\square^p$, temporal queries are classified into snapshot-reducible and extended snapshot-reducible. For the snapshot-reducible queries, $\square^p$ can be propagated without requiring the contents of base relations. However, as delta values are missing (or partially available), such delta propagation is not *net-effect*. For the extended snapshot-reducible queries, as the starting-point of TI in $\square^p$ is missing, it is impossible to evaluate temporal predicates. In consequence, $\square^p$ are translated into $\square$ using base relations in DW. After partial delta transformation is done, incremental procedures described in Section 8 can be used.

Chapter 10

Conclusion and outlook

In this chapter, we first summarize the work and contributions of this thesis. Then, we give an outlook on the related issues that are left open.

10.1 Conclusion and summary

NoSQL databases emerge with the wave of Web 2.0. They claim to handle large amount of data with support of elastic scalability, high availability, high throughput, sophisticated data distribution and replication, schema flexibility and tunable data consistency.

Wide-column store (WCS) is one type of NoSQL databases. It organizes the data in a structured way. In contrast to the traditional database systems (RDBMSs), WCSs introduce a new concept called "column-family" besides the notion of row, column and table. The "column-family" can be seen as a prefix of columns and it denotes all the columns which belong to the same column-family will be stored contiguously on disk. Moreover, each column can store multiple data versions with their corresponding timestamps (TSs).

In recent years, there is a trend to build data warehouses (DWs) based on NoSQL databases. Besides storing the current data, DW also maintains the temporal and time-related data. To keep the state of DW up-to-date, DW frequently extracts data changes from various data sources and incrementally applying such changes to the data warehouse. Although both the incremental DW maintenance and temporal data management and processing have been the subjects of extensive research, no such capabilities have been explored systematically for today's WCSs.

In this thesis, we summarize our findings and proposals over the past years of research which mainly focuses on the following two aspects:

1. How to effectively manage and process temporal data in wide-column stores.
2. How to incrementally re-compute materialized temporal computation results in wide-column stores.

We started with a short description of WCS's history and proposed how temporal data can be stored and processed in the context of WCS. Then, we described how temporal data results (based on various temporal data models and temporal operator models) can be incrementally recomputed.

Temporal data modeling with WCSs

Although each column in a WCS table can maintain multiple data versions, its implicit temporal interval (TI) representation can cause wrong or misleading temporal query results. In consequence, we proposed two alternative table representations, namely, *tuple-timestamping* representation (TTR) and *explicit-history* representation (EHR). TTR appends each tuple with an explicit TI where EHR attaches TI to each data version.

Temporal data processing with WCSs

To analyze data in wide-column stores, users can either write low-level programs, such as MapReduce procedures or utilize high-level languages, such as Pig Latin or Hive. Nevertheless, all of these approaches require users to explicitly implement the desired semantics of temporal query processing. Hence, two temporal operator models (based on the previous two temporal table representations), i.e. TTRO and CTO are defined to facilitate the temporal query specifications. For processing TTR tables, we extended the temporal relational algebra to the $TTRO$ operator model (based on the data model of WCS). For the EHR tables, a new temporal data model called CTO was proposed. Both $TTRO$ and CTO models contain 8 operators, namely, *Union*, *Difference*, *Intersection*, *Project*, *Filter*, *Cartesian product*, *Theta-Join* and *Group by* with a set of aggregation functions, such as SUM, AVG, MAX and etc. The performance comparison has shown that the CTO operator model outperforms the $TTRO$ operator model for the temporal query processing.

Temporal change-data capture

To incrementally maintain the temporal query results in a WCS, we first described how temporal delta can be logically modeled. Due to various temporal delta models utilized in WCSs, a delta item can be represented using row-level-representation or attribute-level-representation. When the attribute-level-representation is used, a tuple reconstruction is needed for certain incremental applications, e.g. a filter operation. To achieve that, we could either let the applications rebuild the row or append the unchanged data columns to the delta output (enhanced attribute-level-representation). The drawback of the first strategy is that it increases the processing overhead of incremental applications. Although the second approach facilitates the data processing task, it increases the delta extraction time and disk usage. To extract temporal deltas from WCSs, five CDC approaches, namely, *Timestamp-based approach*, *Audit-column approach*, *Log-based approach*, *Trigger-based approach* and *Snapshot differential approach* can be used.

For comparing the performance of different CDC approaches, we considered three different delta formats. For all three, the Trigger-based approach is always the best option when the data modification rate is low. When the data modification rate is high and CDC uses the attribute-level-representation, the Log-based approach is faster than the other approaches. However, if the CDC utilizes the row-level-representation and enhanced attribute-level-representation, the Timestamp-based approach is faster than the Log-based approach. Nevertheless, even though the Log-based approach is slightly slower at a high data modification rate, it is still preferable as it can generate a complete delta history.

Incremental temporal delta propagation with complete temporal deltas

As CDC approaches can produce temporal deltas at different completeness level, namely, not all the CDC approaches can generate a complete temporal delta set, we have described the incremental temporal change data propagation rules for the complete temporal deltas and the partial temporal deltas, respectively.

In contrast to representing the complete temporal deltas as insertions and deletions, we represent the complete temporal deltas as insertions Δ and TI-updates $\square$. As a WCS table usually maintains a large number of columns,

treating update as an individual class can be more efficient when only a small number of columns are modified/updated.

However, when propagating $\square$, it can cause deletions which lead to delta propagation *unclosed*. In consequence, we classified temporal queries as "snapshot-reducible" and "extended snapshot-reducible". To maintain the snapshot-reducible queries, an update can be propagated as update itself. For the extended snapshot-reducible queries, an update needs to be propagated as a deletion/insertion pair.

For propagating temporal deltas, row-level-representation and enhanced attribute-level-representation are used for $TTRO$ and CTO operator models, respectively. At a highly abstract level, both represent deltas in a row-oriented way. In consequence, the change data propagation rules proposed for $TTRO$ operators are also feasible for CTO operators.

However, for CTO operators, as the enhanced attribute-level-representation contains the name of modified column in row key, we can optimize the incremental algorithms by first checking whether the modified columns are referenced in the projection attributes or predicates. Moreover, as a version-modification can cause either a version-modification or modifications of multiple versions, we classified the types of temporal view modification as single version-insertion I_v, single version-update μ_v, single version-deletion D_v, multi-version insertions I_{mv}, multi-version updates μ_{mv} and multi-version deletions D_{mv}.

To incrementally maintain the temporal aggregation, we explored a data structure called *inverted index*. Each inverted index can be stored as a normal WCS table and no additional modifications of WCSs are needed. Comparing to the existed approaches, maintaining inverted index is straightforward and easier.

Incremental temporal delta propagation with partial temporal deltas

For partial temporal delta propagation, we have indicated that directly propagating partial temporal delta (without the partial temporal delta transformation) is impossible. In consequence, partial temporal deltas are transformed into the complete temporal deltas. During delta transformation, upsert and partial update deltas will be treated as insertions Δ where the values of TI-updates $\square$ are partially available. To propagate partial TI-update deltas $\square^p$, temporal queries are classified into snapshot-reducible and extended snapshot-reducible. For the snapshot-reducible queries, $\square^p$ can be

propagated without requiring the contents of base relations. However, as delta values are missing (or partially available), such delta propagation is not *net-effect*. For the extended snapshot-reducible queries, as the starting-point of TI in $\square^p$ is missing, it is impossible to evaluate temporal predicates. Hence, $\square^p$ are translated into $\square$ using the base relations in DW. After partial delta transformation is done, incremental procedures described for the complete deltas can be used.

10.2 Outlook and future work

Although this thesis has extensively studied the issues of temporal data management and incremental data recomputation with Wide-column stores (WCSs) and MapReduce, it is impossible to cover and discuss all the related topics. In consequence, we describe some untouched topics which are left open for future work and improvements.

Temporal data modeling

In our thesis, two table representations are proposed to store temporal data in WCSs. In our consideration, we assume that the users or applications are aware of the correctness of temporal data.

However, when applications depend on WCSs to guarantee the data correctness, more system-supports need to be added. For example, users are able to specify temporal integrity constraints when issuing the *create table* command.

Enhanced temporal operators

We defined two temporal operator models in this thesis and each of them contains 8 temporal operators. In general, these temporal operators are able to support basic query specifications and various aggregations which are sufficient to run temporal benchmark.

Nevertheless, to support more complex queries, e.g. TOP operation and partial temporal join, more enhanced temporal operators need to be added.

Flexible schema support

One of the main benefits supported by WCSs is schema flexibility, namely, users can modify the table schema on the fly. However, it is very hard to give a formal definition of an operator based on schema flexibility, as the domains of tables vary from time to time and hence the *set* theory is broken.

Although several proposals address such problems, e.g. Pig Latin [PUn] requires users to explicitly specify the input and output schemata of *Union* operator, there is no good solution until now.

Query implementation and optimization

In our experiment, each temporal operator is implemented by utilizing the MapReduce Framework and the optimization of temporal query execution is done manually, e.g. predicate push-down.

In the future work, we can replace the MapReduce framework by another popular data parallel processing frameworks, e.g. *Spark* [Spa]. Comparing to MapReduce, Spark will not store the intermediate results on disk and can hence increase the query performance. For optimizing the temporal query execution, we can build secondary index to accelerate the processing of temporal filter and temporal join. Moreover, a cost-based model can be built for the temporal query optimization.

Incremental recomputation with valid time

In this thesis, we mainly focused on the *transaction time*, namely, time information is automatically generated by a WCS. Although our temporal operator models do not depend on the semantics of timestamp, CDC approaches and the corresponding CDC outputs can be heavily affected. For example, when users can arbitrarily modify the time information, an update occurred in WCSs can overwrite the old value. In consequence, the timestamp of each data version does not indicate the time of data modification anymore and Timestamp-based approach is infeasible.

Data consistency

In our implementation, "HBase" is adopted as the data storage layer. It sacrifices system availability to support data consistency and partition tolerance.

However, for the other WCSs, e.g. "Cassandra", it chooses to support partition tolerance and system availability and sacrifices data consistency. In consequence, when Cassandra is utilized as the storage layer, a suitable data consistency model needs to be built.

Bibliography

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, Nov. 1983.

[BBJ] Michael H. Böhlen, Renato Busatto, and Christian S. Jensen. Point- versus interval-based temporal data models.

[Bea95] L. Bakgaard and et. al. Incremental computation of time-varying query expressions. In *IEEE Transactions*, pages 583–590, 1995.

[Ber] Oracle Berkeley DB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`.

[BkLLT86] Jose A. Blakeley, Per Åke Larson, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. pages 61–71, 1986.

[BOea98] Ohlen Busatto, M. H. B Ohlen, and et al. Point- versus interval-based temporal data models, 1998.

[Bre12] Eric Brewer. Cap twelve years later: How the "rules" have changed. *IEEE Explore*, 45:23–29, 2012.

[BSS96] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. Coalescing in temporal databases. *IEEE Computer*, 19:35–42, 1996.

[BWR+11] Pramod Bhatotia, Er Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: Mapreduce for incremental

computations. In *In Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[Cas] Cassandra. `https://cassandra.apache.org/`.

[CC87] James Clifford and Albert Croker. The historical relational data model (HRDM) and algebra based on lifespans. *Proceedings of the Third International Conference on Data Engineering*, pages 528–537, 1987.

[CDea06] Fay Chang, Jeffrey Dean, and et al. Bigtable: A distributed storage system for structured data. *OSDI'06*, pages 205–218, 2006.

[CDea08] Fay Chang, Jeffrey Dean, and et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[Cod70] F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, pages 377–387, 1970.

[Com08] Werner Vogels Amazon. Com. Eventually consistent, 2008.

[Cou] CouchDB. `http://couchdb.apache.org/`.

[CSN] Strozzi NoSQL. `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL`.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. *SoCC*, 2010.

[DBG12] Anton Dignoes, Michael H. Boehlen, and Johann Gamper. Temporal alignment. *SIGMOD*, pages 433–444, 2012.

[DBS96] Debabrata Dey, Terence M. Barron, and Veda C. Storey. A complete temporal relational algebra. *VLDB Journal*, 5:5–167, 1996.

[Dea04] Jeffrey Dean and et al. Mapreduce: Simplified data processing on large clusters, 2004.

[DL02] H. D. J. Date and N. Lorentzos. *Temporal Data and the Relational Model*. Morgen Kaufmann, 2002.

[DMGG15] Andrea De Mauro, Marco Greco, and Michele Grimaldi. What is big data? a consensual definition and a review of key research topics. *AIP Conference Proceedings*, 1644:97–104, 2015.

[DS08a] David J. DeWitt and Michael Stonebraker. Mapreduce: A major step backwards - part 1. `http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/`, 2008.

[DS08b] David J. DeWitt and Michael Stonebraker. Mapreduce: A major step backwards - part 2. `http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/`, 2008.

[EFH+11] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer, and Markus Brückner. *NoSQL Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 2011.

[Fvs] The Four V's of Big Data. `http://www.ibmbigdatahub.com/infographic/four-vs-big-data`.

[Gea95] Timothy Griffin and et al. Incremental maintenance of views with duplicates, 1995.

[Geo11] Lars George. *HBase: The Definitive Guide*. O'REILLY, 2011.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, pages 51–59, 2002.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications, 1995.

[GR09] Matteo Golfarelli and Stefano Rizzi. A survey on temporal data warehousing. *International Journal of Data Warehousing and Mining (IJDWM)*, 2009.

[Hba] HBase. `https://hbase.apache.org/`.

[HD13] Yong Hu and Stefan Dessloch. Extracting deltas from column oriented nosql databases for different incremental applications and diverse data targets. *Advances in Databases and Information Systems*, 642:372–387, 2013.

[HD14a] Yong Hu and Stefan Dessloch. Defining temporal operators for column oriented nosql databases. *Advances in Databases and Information Systems*, 8716:39–55, 2014.

[HD14b] Yong Hu and Stefan Dessloch. Extracting deltas from column oriented nosql databases for different incremental applications and diverse data targets. *Journal of Data and Knowledge Engineering*, 2014.

[HD15a] Yong Hu and Stefan Dessloch. Incrementally maintaining materialized temporal views in column-oriented nosql databases with partial deltas. *Advances in Databases and Information Systems*, 539:88–96, 2015.

[HD15b] Yong Hu and Stefan Dessloch. Temporal data management and processing with column oriented nosql databases. *Journal of Database Management*, 2015.

[HDH16] Yong Hu, Stefan Dessloch, and Klaus Hofmann. Temporal view maintenance in wide-column stores with attribute-timestamping model. *Advances in Databases and Information Systems*, Accepted, 2016.

[Hew10] Eben Hewitt. *Cassandra: The Definitive Guide*. O'REILLY, 2010.

[Hiv] Hive. `http://hive.apache.org/`.

[HQ12] Yong Hu and Weiping Qu. Efficiently extracting change data from column oriented nosql databases. *ICS*, 2:587–598, 2012.

[HWCL14] Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. Toward scalable systems for big data analytics: A technology tutorial. *Access, IEEE*, 2:652–687, 2014.

[Jac09] Adam Jacobs. The pathologies of big data. *ACM queue*, 7(6), July 2009.

[JD08] Thomas Jörg, , and Stefan Dessloch. Towards generating etl processes for incremental loading. *IDEAS*, pages 101–110, 2008.

[JD09] Thomas Jörgand and Stefan Dessloch. Formalizing etl jobs forincremental loading of data warehouses. *BTW*, pages 327–346, 2009.

[JD11] Thomas Jörg, , and Stefan Dessloch. View maintenance using partial deltas. *BTW*, pages 287–306, 2011.

[JG09] et al. J. Gampe. Temporal aggregation. *Encyclopedia of Database Systems*, pages 2924–2929, 2009.

[JMRS93] Christian S. Jensen, Leo Mark, Nick Roussopoulos, and Timos Sellis. Using differential techniques to efficiently support transaction time. *The VLDB Journal*, 2:75–111, 1993.

[JMS95] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model (extended abstract), 1995.

[JMS09] Thomas Joerg, Albert Maier, and Oliver Suhre. Generating extract, transform, and load (etl) jobs for loading data incrementally. *United States Patent*, 2009.

[Jör13] T. Jörg. *Incremental Recomputations in Materialized Data Integration*. 2013.

[JPYD11] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in mapreduce. In *Proceedings of the Third International Workshop on Cloud Data Management*, CloudDB '11, pages 7–14. ACM, 2011.

[JSS] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. Unifying temporal data models.

[JSS94] Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. The tsql2 data model, 1994.

[KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, Crosspoint Boulevard, Indianapolis, 2004.

[KeM11] Krishna Kulkarni and Jan eike Michels. Temporal features in sql:2011. *SIGMOD*, pages 34–43, 2011.

[KFea14] Martin Kaufmannyx, Peter M. Fischer, and et. al. Benchmarking databases with history support, 2014.

[KM12] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[KS95] N. Kline and R. Snodgrass. Computing temporal aggregations. *Data Eng.*, pages 222–231, 1995.

[LGM96a] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. *VLDB*, pages 63–74, 1996.

[LGm96b] Wilburt Juan Labio and Hector Garcia-molina. Efficient snapshot differential algorithms for data warehousing. In *In Proceedings of the International Conference on Very Large Data Bases*, pages 63–74, 1996.

[LLea] Rubao Lee, Tian Luo, and et. al. Facebook data infrastructure team.

[LM97] Nikos A. Lorentzos and Yannis G. Mitsopoulos. Sql extension for interval data. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):480–499, May 1997.

[LMSS95] James Lu, Guido Moerkotte, Joachim Schü, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *In SIGMOD*, pages 340–351, 1995.

[LO09] L. Liu and T. M. Oezsu. *Encyclopedia of Database Systems*. Springer Verlag, 2009.

Bibliography

[LOR+10] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 51–62. ACM, 2010.

[LYea00] Wilburt Labio, Jun Yang, and et al. Performance issues in incremental warehouse maintenance. *VLDB*, pages 461–472, 2000.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. *VLDB*, pages 447–453, 1977.

[ML09] Roger Magoulas and Ben Lorica. Introduction to big data. *O'Reilly Media*, (11), February 2009.

[Mon] MongoDB. `https://www.mongodb.org/`.

[MQea97] Inderpal Singh Mumick, Dallan Quass, and et al. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD*, pages 100–111, 1997.

[MZ15] Elzbieta Malinowski and Esteban Zimanyi. Temporal data warehouses. *Advanced Data Warehouse Design*, pages 185–249, 2015.

[Nosa] Nosql databases. `http://nosql-database.org/`.

[NoSb] NoSQL Meeting. `http://www.eventbrite.com/e/nosql-meetup-tickets-341739151`.

[ORea] Christopher Olston, Benjamin Reed, and et. al. Pig latin: A not-so-foreign language for data processing.

[PBYI09] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. Dryadinc: Reusing work in large-scale computations. *HotCloud 09*, 2009.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Pig] Pig. `http://pig.apache.org/`.

[PPRea09] Andrew Pavlo, Erik Paulson, Alexander Rasin, and et. al. A comparison of approaches to large-scale data analysis. In *In SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[PUn] Pig reference book. `http://pig.apache.org/docs/r0.10.0/basic.html#union`.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3:337–341, 1991.

[Red] Redis. `http://redis.io/`.

[Ria] Riak. `http://basho.com/products/#riak`.

[RW12] Eric Redmond and Jim R. Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. 2012.

[RWE11] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Database*. O'REILLY, 2011.

[SAea94] Richard T. Snodgrass, Ilsoo Ahn, and et al. Tsql2 language specification. *SIGMOD Record*, 23:65–86, 1994.

[SBea97] Richard T. Snodgrass, Michael H. Boehlen, and et. al. Transitioning temporal support in tsql2 to sql3, 1997.

[SF12] PramodkumarJ. Sadalage and Martin Fowler. *NoSQL Distilled*. Addison Wesley, 2012.

[SGM93] R. Snodgrass, S. Gomez, and L. Mckenzie. Aggregates in the temporal query language tquel. *Data Eng.*, pages 826–842, 1993.

[SJHD14] J. Schildgen, T. Jorg, M. Hoffmann, and S. Dessloch. Marimba: A framework for making mapreduce jobs incremental. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 128–135, June 2014.

[Spa] Spark. `spark.apache.org`.

[Tan97] A. U. Tansel. Temporal relational data model. *IEEE transaction on knowledge and data engineering*, 9:464–479, 1997.

[TG89] A. U. Tansel and L. Garnett. Nested historical relations. *SIGMOD*, 18:284–294, 1989.

[THC09] et al. Thomas H. Cormen. *Introduction to Algorithms*. MIT Press, 2009.

[TSea09] Ashish Thusoo, Joydeep Sen Sarma, and et. al. Hive- a warehousing solution over a map-reduce framework. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, pages 1626–1629, 2009.

[TSea10] Ashish Thusoo, Joydeep Sen Sarma, and et. al. Hive - a petabyte scale data warehouse using hadoop. In *In Proceedings of International Conference on Data Engineering (ICDE*, pages 996–1005, 2010.

[Tum92] P. Tuma. Implementing historical aggregates in tempis. *Mater's thesis, Wayne State University*, 1992.

[Uni] Villanova Universtiy. What is big data. `http://www.villanovau.com/resources/bi/what-is-big-data/`.

[UW97] Jeffrey Ullman and Jennifer Widom. *A First Course In Database Systems*. Prentice Hall, 1997.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'REILLY, 2012.

[YCS] YCSB. `https://github.com/brianfrankcooper/YCSB`.

[YW98] Jun Yang and Jennifer Widom. Maintaining temporal views over non-temporal information sources for data warehousing. In *In Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 389–403, 1998.

[YW00] Jun Yang and Jennifer Widom. Temporal view self-maintenance in a warehousing environment. In *In Proc. of the 2000 Intl. Conf. on Extending Database Technology*, pages 395–412, 2000.

[YW03] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB Journal*, pages 262–283, 2003.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *IN PROCEEDINGS OF SIGMOD*, pages 316–327, 1995.

Curriculum Vita

Personal information

First Name	Yong
Last Name	Hu
Birthplace	Wuhan, China

Education

1/2012 – 12/2015	Ph.D candidate, Heterogeneous Information Groups, University of Kaierslautern
4/2009 – 11/2011	Graduate Study Informatik, University of Kaierslautern
09/2006 – 11/2009	Master in School of Software Engineering, Huazhong University, China
09/2002 – 07/2006	Bachelor in Computer Science, Huazhong University, China

Work Experience

since 02/2016	Software Engineer, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaierslautern
05/2007 – 03/2008	Software developer for LDAP, member of Siemens Cooperatives Laboratory group, Huazhong University, China
05/2007 – 03/2008	Software developer for Real-time Ecommerce, member of Siemens Cooperatives Laboratory group, Huazhong University, China
07/2005 – 09/2005	Internship, Network Center of Wuhan Union Hospital, China