TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

PHD DISSERTATION

On-Demand ETL for Real-Time Analytics

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Dipl.-Inf. Weiping Qu

Date of Defense:
June 9, 2020
Dean:
Prof. Dr. Jens Schmitt
Reviewer:
Prof. Dr.-Ing. Stefan Deßloch
Prof. Dr.-Ing. Bernhard Mitschang

D 386

For my father, mother, parents-in-law,
my wife and my daughter(s)

Acknowledgements

This dissertation summarizes my research work during the period when I worked as a member of the Heterogeneous Information Systems (HIS) group at the University of Kaiserslautern while it was finished in the past two years when I am working in the Online Computing group at Ant Financial in Shanghai, China after I returned back from Germany. I think I am fortunate that the 5-year research experience helped me to successfully take the fist step towards industry while the most significant two-year working experience in turn adds more *nutrition* to this dissertation.

First, I would like to thank my advisor, Prof. Dr.-Ing Stefan Deßloch for offering me the opportunity of pursuing my research work in the most interesting area in the world: database and information systems. His guidance on the research and teaching work helped me to complete my view in scientific research in a more systematic manner. In addition to work, I want to express my deepest gratitude to him, as he delivered strong support when I was facing my family issues during that tough time and also encouraged and guided me to finish this dissertation when I was struggling with my work at Ant Financial.

Next, I would like to thank Prof. Dr.-Ing. Bernhard Mitschang for taking the role of the second examiner and Prof. Dr. Christoph Garth for acting as chair of the PhD committee.

Furthermore, I would like to thank my first *mentor* Dr. Tianchao Li, who guided me with my diploma thesis at IBM Böblingen and opened up the door of database research for me. Besides, I want to thank Prof. Dr.-Ing. Dr.h.c. Theo Härder who shows his great passion in the database research area and serves as a great example to the young generations. Moreover, I am also grateful to Prof. Dr.-Ing Sebastian Michel for his interesting thoughts and valuable discussions.

I am grateful to my master thesis students: Nikolay Grechanov, Dipesh Dangol, Sandy Ganza, Vinanthi Basavaraj and Sahana Shankar, who also contributed their efforts to this dissertation with their solid knowledge and highly motivated attitude.

I am indebted to my former colleagues and friends, especially Dr. Thomas Jörg, Dr. Yong Hu, Dr. Johannis Schildgen, Dr. Sebastian Bächle, Dr.

Yi Ou, Dr. Caetano Sauer, Dr. Daniel Shall, Dr. Kiril Panev, Dr. Evica Milchevski, Dr. Koninika Pal, Manuel Dossinger, Steffen Reithermann and Heike Neu, with whom I had a great time in Kaiserslautern.

Also, I would like to thank my best friends, Xiao Pan, Dr. Yun Wan and Yanhao He, who always have interesting ideas and made every day of these years in Kaiserlautern sparkling.

Finally, I would like to express my love to my family who always supports me behind the curtain. I want to thank my parents, She Qu and Xingdong Zhang, who educated me the most strictly and love me the most deeply. Without them, I would never have gone that far. I want to thank my wife, Jingxin Xie, who always stands beside me and encourages, motivates and supports me towards the final success of my PhD carrier, and also thank her for bringing our precious daughter to this world, who really brings our life to a brand new world and makes it enjoyable.

Weiping Qu
Shanghai, Jan 2020

Abstract

In recent years, business intelligence applications become more real-time and traditional data warehouse tables become fresher as they are continuously refreshed by streaming ETL jobs within seconds. Besides, a new type of federated system emerged that unifies domain-specific computation engines to address a wide range of complex analytical applications, which needs streaming ETL to migrate data across computation systems.

From daily-sales reports to up-to-the-second cross-/up-sell campaign activities, we observed various latency and freshness requirements set in these analytical applications. Hence, streaming ETL jobs with regular batches are not flexible enough to fit in such a mixed workload. Jobs with small batches can cause resource overprovision for queries with low freshness needs while jobs with large batches would starve queries with high freshness needs. Therefore, we argue that ETL jobs should be self-adaptive to varying SLA demands by setting appropriate batches as needed. The major contributions are summarized as follows.

- We defined a consistency model for "On-Demand ETL" which addresses correct batches for queries to see *consistent* states. Furthermore, we proposed an "Incremental ETL Pipeline" which reduces the performance impact of on-demand ETL processing.
- A distributed, incremental ETL pipeline (called HBelt) was introduced in distributed warehouse systems. HBelt aims at providing consistent, distributed snapshot maintenance for concurrent table scans across different analytics jobs.
- We addressed the elasticity property for incremental ETL pipeline to guarantee that ETL jobs with batches of varying sizes can be finished within strict deadlines. Hence, we proposed Elastic Queue Middleware and HBaqueue which replace memory-based data exchange queues with a scalable distributed store - HBase.
- We also implemented lazy maintenance logic in the extraction and the loading phases to make these two phases workload-aware. Besides, we discuss how our "On-Demand ETL" thinking can be exploited in analytic flows running on heterogeneous execution engines.

Contents

1 Introduction **1**
- 1.1 State-of-the-art ETL . . . 1
- 1.2 Motivation . . . 4
  - 1.2.1 Working Scope . . . 4
  - 1.2.2 Problem Analysis . . . 5
  - 1.2.3 On-Demand ETL . . . 7
- 1.3 Research Issues . . . 8
- 1.4 Outline . . . 10

2 Preliminaries **13**
- 2.1 Incremental View Maintenance . . . 13
- 2.2 Near Real-time Data Warehouses . . . 14
  - 2.2.1 Incremental ETL . . . 15
  - 2.2.2 Near real-time ETL . . . 15
- 2.3 Lazy Materialized View Maintenance . . . 16
  - 2.3.1 Right-time ETL . . . 17
  - 2.3.2 Relaxed Currency in SQL . . . 18
- 2.4 Real-time Data Warehouse . . . 19
  - 2.4.1 One-size-fits-all Databases . . . 19
  - 2.4.2 Polystore & Streaming ETL . . . 20
  - 2.4.3 Stream Warehouse . . . 20
- 2.5 Streaming Processing Engines . . . 21
  - 2.5.1 Aurora & Borealis . . . 21
  - 2.5.2 Apache Storm . . . 23
  - 2.5.3 Apache Spark Streaming . . . 25
  - 2.5.4 Apache Flink . . . 27
  - 2.5.5 S-store . . . 29
  - 2.5.6 Pentaho Kettle . . . 30
- 2.6 Elasticity in Streaming Engines . . . 30
- 2.7 Scalable NoSQL Data Store: HBase . . . 32
  - 2.7.1 System Design . . . 32
  - 2.7.2 Compaction . . . 34
  - 2.7.3 Split & Load Balancing . . . 34
  - 2.7.4 Bulk Loading . . . 35

3 Incremental ETL Pipeline **37**
3.1 The Computational Model 37
3.2 Incremental ETL Pipeline 39
3.3 The Consistency Model 41
3.4 Workload Scheduler 43
3.5 Operator Thread Synchronization and Coordination 46
3.5.1 Pipelined Incremental Join 46
3.5.2 Pipelined Slowly Changing Dimensions 49
3.5.3 Discussion 53
3.6 Experiments 53
3.6.1 Incremental ETL Pipeline Performance 54
3.6.2 Consistency-Zone-Aware Pipeline Scheduling . . . 56
3.7 Summary 58

4 Distributed Snapshot Maintenance in Wide-Column NoSQL Databases **61**
4.1 Motivation 61
4.2 The HBelt System for Distributed Snapshot Maintenance . 63
4.2.1 Architecture Overview 63
4.2.2 Consistency Model 65
4.2.3 Distributed Snapshot Maintenance for Concurrent Scans 68
4.3 Experiments 70
4.4 Summary 74

5 Scaling Incremental ETL Pipeline using Wide-Column NoSQL Databases **75**
5.1 Motivation 75
5.2 Elastic Queue Middleware 77
5.2.1 Role of EQM in Elastic Streaming Processing Engines 77
5.2.2 Implementing EQM using HBase 79
5.3 EQM Prototype: HBaqueue 82
5.3.1 Compaction 82
5.3.2 Resalting 83
5.3.3 Split Policy 84
5.3.4 Load balancing 85
5.3.5 Enqueue & Dequeue Performance 89
5.4 Summary 90

6 Workload-aware Data Extraction **93**
6.1 Motivation 93

6.2 Problem Analysis 96
6.2.1 Terminology 96
6.2.2 Performance Gain 100
6.3 Workload-aware CDC 100
6.4 Experiments 104
6.4.1 Experiment Setup 104
6.4.2 Trigger-based CDC Experiment 104
6.4.3 Log-based CDC Experiment 105
6.4.4 Timestamp-based CDC Experiment 107
6.5 Summary 107

7 Demand-driven Bulk Loading 109
7.1 Motivation 109
7.2 Architecture 110
7.2.1 Offline Loading Index Construction 112
7.2.2 Online Loading and Query Coordination 114
7.3 Experiments 115
7.3.1 Experiment Setup 116
7.3.2 Loading Phase 116
7.3.3 On-demand Data Availability 118
7.4 Summary 120

8 Real-time Materialized View in Hybrid Analytic Flow 121
8.1 Motivation 121
8.2 Incremental Recomputation in Materialized Integration . . 122
8.3 Real-time Materialized Views in Hybrid Flows 124
8.3.1 Performance Gain 127
8.3.2 View Selection and Deployment 129
8.4 Experiments 132
8.4.1 View Selection on Single Platform 133
8.4.2 View Deployment in Hybrid Flows 135
8.5 Summary 136

9 Conclusion and Future Work 139
9.1 Conclusion 139
9.2 Outlook 140

List of Figures 143

List of Tables 147

Bibliography 149

1 Introduction

1.1 State-of-the-art ETL

ETL is a process that *extracts* users' transaction data from primary sources, *transforms* it and *loads* it in a separate, dedicated warehousing system for business analytics [KC04]. ETL has evolved over several decades and now a new generation of ETL has emerged which absorbs streaming processing capabilities to continuously refresh data warehouses (DWs) for real-time analytics.

Traditional ETL jobs were performed offline and processed daily batches over night, which enables business analysts to generate business reports for decision making on the other day. Traditional ETL consists of three stages: extract, transform and load, and each stage takes daily-generated files as input, processes them and generates output files for the next stage. High throughput is thereby achieved, which guarantees that such daily-batch ETL jobs could be finished within hours over night.

Near real-time ETL [VS09, JD09b] was later introduced for online data warehousing. ETL jobs take only changes derived from source tables and generate final deltas to update target warehouse tables incrementally. The batch was shortened to micro-batches (e.g., several minutes), which significantly reduces the execution latency and the time window. Business analysts could have reports derived from fresher DW tables in data warehousing systems where the incremental ETL jobs and the executions of Online Analytical Processing (OLAP) queries are scheduled at runtime.

A wave of research work emerged afterwards, which focused on acceleration of OLAP queries for real-time analytics by optimizing data management on new hardware. As analytical workload is mainly characterized as read-only table-scans, data is stored in columnar layout and mainly stored in memory for efficient access. In addition, novel technologies on new hardware like multi-core, GPU and FPGA were used to accelerate table scans with their parallel processing features. These techniques opened up a new scenario of delivering up-to-date analytical results to users for OLAP queries directly in the same platform as for original Online Transaction Processing (OLTP) [KN11, FCP+12] . Another alternative was to

integrate read-optimized OLAP engines with existing OLTP systems. For example, IBM Data Analytics Accelerator [BBD+15] enhances IBM DB2 for z/OS with additional Business Intelligence (BI) processing capabilities. By offloading OLTP data from original DB2 tables to accelerator-specific tables, reporting processes or complex model training applications can be performed directly over current operational data using the read-optimized Netezza appliance, thus increasing business values. However, the time gap between source transactions and analysis-ready data still cannot be ignored and needs to be addressed by ETL.

In order to exploit business values and approach real-time analytics, a lot of research efforts have been carried out in the area of streaming ETL to reduce the latency of cumbersome ETL jobs.

Simitsis et al., [DCSW09, SWDC10, SWCD12, KVS13] believe that today's BI should be operational BI, in which case the data that is cleansed with high quality in the DWs should flow back to operational databases to increase business values. For example, online retailers would adjust the product discounts based on up-to-the-minute customer profiles in a certain campaign. This requires ETL processes to be more general data flows so that optimizations could be carried out from end to end according to different quality metrics (performance, fault tolerance, freshness, etc.) demanded by customers. For instance, to have fresher data for analysis, more frequent, shorter refresh cycles are used to stream new updates from sources to the DW. The key behind this scenario is that ETL processes should have the capability of adapting to different customer needs and varying workloads.

Golab et al., [GJSS09, GJS11] mainly focused on their research work in streaming DWs, which combine the features of traditional DWs and data stream systems and mainly serve applications like online stock trading, credit card fraud detection and network monitoring. In their setting, ETL jobs are triggered immediately, as new data arrive from source data streams. Based on the *data staleness* metric defined for target warehouse tables, ETL jobs are scheduled for minimum average staleness under resource control. They also introduced *temporal consistency* for quantifying and monitoring the data quality in their real-time streaming warehouse.

The BigDAWG [EDS+15] is a polystore system which unifies multiple storage engines using federated database system techniques. It is guided by the *one-size-does-not-fit-all* principle [SC05]. As each storage engine is optimized for its specific workload, BigDAWG is capable of handling a variety of specific tasks. In order to efficiently migrate data across different storage engines at runtime, Meehan et al., [MAZ+17] proposed streaming ETL to address lower execution latency for data ingestion and da-

ta transformation. Streaming ETL is provided by a transactional stream processing system called S-Store, which guarantees ordered execution, exactly-once processing and ACID properties when accessing shared mutable states. More specifically, each unbounded input stream is cut into *atomic batches* of fixed sizes, which are further processed on each processing operator of an ETL job graph in a transactional manner.

The aforementioned streaming ETL solutions have their limitations in specific scenarios. However, in recent decades, business companies are migrating from traditional business intelligence (BI) to Big Data era. With the famous 4Vs: *Volume*, *Velocity*, *Variety* and *Value* [VOE11, ZE$^+$11, R$^+$11], analytics over Big Data forces the evolution of the underlying computation infrastructure for more business values derived from computations over high-volume data in a timely manner. More specifically, these issues can be seen as follows:

- *Variety*: Different forms of raw data requires extra computation overhead and data transfer cost, which are generally addressed by ETL tools.
- *Velocity*: Reporting on a daily basis nowadays can not add more business values. To rapidly react to market changes, underlying computation engines have to be equipped with real-time processing capability. Stream processing is generally used to address this issue.
- *Volume*: The huge volume is normally handled by distributed processing systems, which also have strong scalability at different data-growth rate.
- *Value*: To be more competitive in current business market, companies are not satisfied with simple daily-report. More timely, accurate, complex data analytics (e.g. machine learning) are heavily used in big companies. Different types of workloads are running in the same data repository for different business requirements.

Google MapReduce (MR) [DG08] and Apache Hadoop [Whi12] were invented to provide batch-processing capability over large-scale source files. A MR job might consist of multiple stages, each of which writes complete outputs to distributed file system to achieve fault-tolerance, i.e., avoiding stragglers and failed nodes.

Aiming at cumbersome MR jobs, Spark [spa] was later introduced to efficiently speed up batch-processing by transferring data across stages through memory instead of reading/writing files. Spark streaming

[ZDL+13] is a streaming solution built over Spark and utilizes the original batch-processing capability for streaming processing. Unbounded streams are split into micro-batches of fixed time windows (e.g., 1 sec) and processed by the same underlying Spark infrastructure.

In contrast to a streaming overlay over batch-oriented platforms, Flink [CKE+15] addressed the same business scenarios whereas it was designed as a pure streaming processing engine. Batch-processing is considered as a special case of streaming processing from the perspective view of Flink. Both Spark streaming and Flink have their significant influences in the market of streaming ETL and real-time Big Data analytics.

1.2 Motivation

Based on a short survey on state-of-the-art ETL trends, we look into problems and challenges of streaming ETL more from the perspective view of business analytics. The diversity of workloads (i.e., ETL jobs, OLAP queries) and different data freshness requirements (e.g., real-time analysis) motivated us to bring ETL and query processing together in an integrated picture when we addressed research issues in the streaming ETL domain.

1.2.1 Working Scope

To better analyze the problem and describe our motivation, we first illustrate a high-level Streaming ETL architecture that sets the scope of our work.

Streaming ETL jobs run continuously and propagate data changes (*deltas*) to the target DWs, where OLAP queries are executed. Multi-version concurrency control techniques are used in DWs to provide OLAP queries and streaming ETL jobs with concurrent access to warehouse tables. During ETL processing, unbounded delta streams are divided into small, bounded, distinct batches. Each time when a delta batch is finished by ETL processing, a new snapshot is created from existing warehouse tables to record the warehouse state. In this way, incoming OLAP queries can be executed over old state independently while concurrent ETL jobs are writing new state at the same time.

More specifically, in our working scope, the streaming ETL jobs consist of Change Data Capture (CDC) processes and Change Data Propagation (CDP) processes. As depicted in Figure 1.1, CDC processes keep

pulling (e.g., crawling transaction logs) up-to-date *deltas* and pushing into so-called *source delta streams* that are allocated in the data staging area used for ETL job processing. The data staging area is used to alleviate the possible overload of the sources and the warehouse. It relieves *traffic jams* at a high delta-insertion rate and meanwhile, at a very high query rate, alleviates the burden of locking caused by concurrent read/write accesses to shared data partitions.

Source delta streams are cut into smaller batches of fixed sizes or fixed time windows. Incremental ETL techniques are applied by the CDP processes on each delta batch to generate a snapshot for ad-hoc OLAP query processing or business reporting. The snapshot is taken at the beginning of a query execution and used during the entire query lifetime without interventions by concurrent updates.

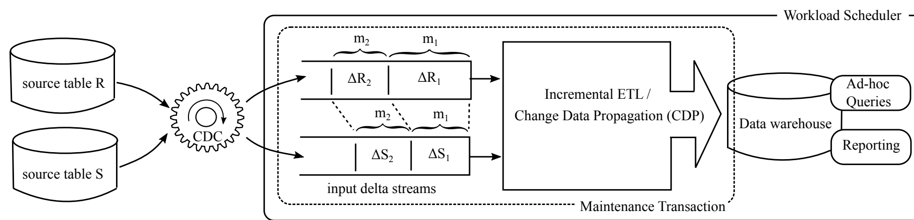


Figure 1.1: The scope of work in this dissertation

1.2.2 Problem Analysis

From daily-sales reports to up-to-the-second cross-/up-sell campaign activities, we have observed a wide range of analytical applications with various latency and freshness demands. Processing streaming ETL jobs in regular micro-batches doesn't seem to be flexible enough to react to the freshness demands of these mixed workloads. If large (minute) batches are used, creating a snapshot for one batch can starve those OLAP queries with high freshness demands, as they have to bear extra cost of slow snapshot generation. If small (second) batches are used, more CPU cycles and memory are needed to process streaming ETL jobs and more redundant storage buffers are required to allow frequent snapshot generation. Especially for queries with low freshness needs, these resources are over-provisioned. It would be beneficial if the underlying streaming ETL infrastructure had the flexibility to align its micro-batch processing with varying freshness demands.

Similar problems have been addressed in several previous work. Zhou et al., [ZLE07] presented their work to *lazily* maintain materialized views

in databases. As DWs can be considered as a special kind of materialized views over data sources, ETL jobs can be seen as the update transactions that refresh the materialized views. In their work, maintenance of a view is postponed until the system has free cycles or the view is referenced by a query. Their ideal case is that the views are maintained entirely using system free cycles at no cost to updates and queries. However, materialized views reside in the same database as source tables while ETL jobs have more complicated situations in all of the three phases, i.e., extract, transform and load.

Thomsen et al., [TPL08] addressed lazy loading of DWs, called *right-time DWs*. At the DW side, a main-memory based *catalyst table* is introduced to ingest outputs of ETL jobs on demand when a query is issued to the DW tables. To answer this query, a view provides the union of the contents of DW tables and this memory table. Depending on the demanding data freshness specified in the query, there is a way to measure how much data is still missing in the memory table and needs to be loaded at the *right* time. A key problem is that this work heavily relies on the main-memory catalyst table, which might not be able to fit in *Big Data* scenarios addressed by tools like Spark and Flink. Another limitation is that their work focuses mainly on the loading phase of ETL, not the entire ETL pipeline. A complete look at each corner of general ETL or some dominating phases (e.g., extract & transformation phases) for right-time ETL could be appealing, especially for more general data integration flow.

Note that, as described in [TPL08], business analysts have the flexibility to specify how accurate the accessed data should be in the memory table for a SQL query. Similar work has already been examined in database management systems. Guo et al., [GLRG04, GLR05] proposed the currency and consistency constraints in SQL queries. For example, **SELECT * FROM authors CURRENCY BOUND 10 MIN** means that the rows from the **authors** table must be at most 10 minutes out of date. With these constraints, users have the freedom to explicitly specify how stale the data is permitted to be to satisfy their queries, which makes it easy for database systems to schedule update transactions and queries. In our working scope, with currency constraints specified in queries, DW systems are able to detect whether the current snapshot is able to meet the freshness demands of incoming OLAP queries or how fresh a new snapshot needs to be to answer a certain query. In addition, a workload scheduler is needed to orchestrate the executions of streaming ETL jobs and OLAP queries to guarantee that the queries would always see consistent snapshots at the right time.

1.2.3 On-Demand ETL

To support mixed workloads with various freshness demands, we present our *on-demand ETL* solution in this dissertation. Generally, a snapshot taken at the warehouse side is stale for time-critical queries, since required source deltas might still be processed by the ETL engine or even still reside in the source systems. In order to meet the freshness requirement, the source deltas need to be merged within the warehouse tables before a snapshot is taken, leading to extra synchronization cost.

Ideally, ETL jobs of large batches are processed when there is no input query or only queries with low freshness needs, while ETL jobs of micro-batches are processed for real-time analytical queries, meeting low latency and high freshness requirements. The cost of having fresh warehouse snapshots (i.e., synchronization delay) can be amortized by high-throughput batch processing during off-peak hours. Assume that the streaming ETL engines can process data batches of different sizes and incoming OLAP queries can be specified with different currency constraints. A workload scheduler is needed to generate streaming ETL jobs of different batch sizes and the resulting snapshots correspond to different constraint groups. Scheduling strategies should be applied in the scheduler to ensure that the average freshness satisfied is the highest while the number of snapshots is the minimum.

Take the same example shown in Figure 1.1. An event of a query arrival at the warehouse side triggers the system to group the current delta tuples in every source delta stream as a delta batch and to construct an ETL (maintenance) job which takes the delta batches as input and perform one run of incremental flow execution. The output delta batch which is produced by this ETL maintenance flow execution is used to refresh the target warehouse tables and then a snapshot is taken for answering this query. The arrival of Q_1 leads to the construction of a maintenance job m_1. The input for m_1 is one delta batch b_1 with the delta tuples from ΔR_1 and ΔS_1 that are extracted from the source transactions committed before the arrival time of Q_1. The query execution of Q_1 is initially suspended and later resumed when relevant tables are refreshed by the output of m_1. With a sequence of incoming queries, a list of chained maintenance jobs are created for ETL flow to process.

1.3 Research Issues

To approach ideal cases of on-demand ETL, the following issues are particularly interesting both from research and practical perspective.

Efficient execution of maintenance job chain. The aforementioned maintenance job chain needs to be executed in response to corresponding OLAP queries. Sequential execution of ETL flow instances can lead to high synchronization delay at a high query rate. Therefore, parallelism needs to be exploited at a certain level of the flow execution to improve performance. Furthermore, general ETL flows contain operations or complex user-defined procedures which read and write shared state in internal or external storage systems. While running different maintenance jobs in parallel, inconsistency may occur due to uncontrolled access to shared state. Job scheduling is thereby needed to prevent reading inconsistent state.

Consistency model for ETL jobs and OLAP queries. In our work, a warehouse snapshot S_i is considered as consistent for an incoming query Q_i if S_i is contiguously updated by output delta batches from preceding maintenance jobs ($m_1 \sim m_i$) before the submission time of Q_i and is not interfered by fast finished succeeding jobs (e.g. m_{i+1}, which leads to non-repeatable read/phantom read anomalies). A formal definition of this consistency model is missing and deployment of such consistency model might have multiple options. One possible solution is to embed timestamps in delta tuples, table partitions and query conditions. During query execution, only tuples with matching timestamps are returned as final results. However, setting explicit timestamps might incur high storage and processing overheads. As another promising alternative, implicit version control of some DW systems can be exploited to ensure consistency. A workload scheduler is needed to schedule the update statements of a maintenance job and the reading statements of OLAP queries.

Distributed snapshot maintenance. Wide-column NoSQL databases are an important class of NoSQL (Not only SQL) databases which scale horizontally and feature high access performance on sparse tables. With current trends towards big DWs, it is attractive to run existing business intelligence/data warehousing applications on higher volumes of data in wide-column NoSQL databases for low latency by mapping multidimensional models to wide-column NoSQL models or using additional SQL add-ons. For examples, applications like retail management can run over integrated data sets stored in big DWs or in the cloud to capture current item-selling trends. Many of these systems also employ Snapshot Isolation (SI) as a concurrency control mechanism to achieve high throughput

for read-heavy workloads. SI works well in a DW environment, as analytical queries can now work on (consistent) snapshots and are not impacted by concurrent ETL maintenance jobs that refresh fact/dimension tables. However, the snapshot made available in the DW is often stale, since at the moment when an analytical query is issued, the source updates (e.g. in a remote retail store) may not have been extracted and processed by the ETL process in time due to high input data volume or slow processing speed. This staleness may cause incorrect results for time-critical decision support queries. To address this problem, snapshots which are supposed to be accessed by analytical queries need to be first maintained by corresponding ETL flows to reflect source updates based on given freshness needs. Snapshot maintenance in this work means maintaining the distributed data partitions that are required by a query. Since most NoSQL databases are not ACID compliant and do not provide full-fledged distributed transaction support, a snapshot may be inconsistently derived when its data partitions are updated by different ETL maintenance jobs.

Elasticity in the distributed ETL pipeline. With continuous updates occurring at the data source side and analytics requests (with different freshness and deadline needs) issued to the DWs, streaming ETL engines need to process different amounts of input data in time windows of various sizes in an elastic manner. With drastically increasing data rates, the in-memory streaming pipes/buffers between processing operators fill up fast, which normally results in back pressure. This back pressure impact would not be resolved until the system first detects where the bottleneck is, determines the scale out degree, lets the deployment manager apply for new resources, spawns threads and finally reroutes data flow. Especially for stateful operators, additional state-shuffling cost is mandatory. To alleviate back pressure, streaming buffers can spill to disk and also scale across cluster nodes which most of the full-fledged scalable data stores have already implemented. Elasticity is a *must* for our distributed ETL pipeline to support mixed workload with various freshness demands.

Workload-aware change data capture. As a crucial phase of general ETL, CDC processes run continuously under efficient setups to deliver up-to-date source deltas/changes. Data changes have to be transferred from sources to data staging area for further executions of ETL jobs with strict deadlines in response to certain accuracy or freshness demands set by OLAP queries. This process competes for resources with original loads running on data sources and harms data source autonomy while analytic workloads could have different freshness demands. Due to unawareness of dynamic freshness demands during initial CDC setups, the CDC requests can either be over-provisioned with resources on data source plat-

forms or starved. Therefore, all the SLAs of incoming analytical queries need to be balanced in a workload-aware manner, instead of always pursuing real-time analytics.

Loading large-scale data sets on demand. Loading large-scale data sets from ETL cluster to DW cluster and meanwhile providing high system uptime is a challenging task. It requires fast bulk import speed. We observed a trade-off between data availability and query response time. Data distributions and access patterns in target DW tables could be fully exploit with hotness and coldness of output data to be loaded.

Revisiting optimization possibility in hybrid flow. Next-generation business intelligence (BI) enables enterprises to quickly react in changing business environments. Increasingly, data integration pipelines need to be merged with query pipelines for real-time analytics from operational data. Newly emerging hybrid analytic flows have been becoming attractive which consist of a set of ETL jobs together with analytic jobs running over multiple platforms with different functionality. Therefore, this motivates us to rethink of potential optimizations in the end-to-end hybrid flows with respect to on-demand ETL.

1.4 Outline

The remainder of this dissertation is organized as follows.

Chapter 2 provides preliminary concepts and techniques that are relevant to this dissertation, mainly in the areas of ETL, (near) real-time DWs and streaming processing techniques.

This dissertation is built upon a list of my research papers published between 2013 and 2017, together with two master theses which were guided by me. They serve as the basis of this dissertation and their contents are re-organized and presented from Chapter 3 to Chapter 8. Their main contributions are outlined as follows.

Based on [QD17d, QBSD15], Chapter 3 explains the definition of a consistency model we proposed for "On-Demand ETL". To guarantee a *consistent* state, a workload scheduler is thereby introduced to schedule both ETL jobs and OLAP queries. Moreover, we proposed an "Incremental ETL Pipeline" for high efficiency and low latency with experimental proof.

In addition, the rest of Chapter 3 addresses potential consistency issues incurred by concurrent operator processing in our incremental ETL pipeline, which was presented in [QD17b].

[QD17a, QSGD15] further extended previous consistency model to adapt to large-scale, distributed analytics, which is presented in Chapter 4.

A distributed variant of our incremental ETL pipeline (called HBelt) was introduced to achieve the same "On-Demand ETL" objective in distributed warehouse systems (e.g., HBase). HBelt aims at providing distributed snapshot maintenance for concurrent table scans in different analytics jobs.

Like other distributed streaming processing systems, we also addressed classic challenges (e.g., back pressure, slow bottleneck operator, elasticity, etc.) in our HBelt system which provides distributed streaming pipelines. In Chapter 5, we presented our proposals of Elastic Queue Middleware and HBaqueue which replace memory-based data exchange queues with scalable distributed stores (e.g., HBase) and make bottleneck operators auto-scale with scaling input queue-shards [QD17c]. This idead was further examined and proved in [Gre17] with experimental results.

In addition to research contributions to the main *transformation* phase for "On-Demand ETL", we also implemented lazy maintenance logic in the *extraction* [Dan17] and the *loading* phases [QD14a] to make these two phases workload-aware (see Chapter 6 and Chapter 7, respectively).

Chapter 8 discusses the details of how our "On-Demand ETL" thinking is exploited in hybrid analytic flows. In [QD14b, Qu13, QRD13], we present our proposal of real-time materialized views.

Finally, we conclude this dissertation in Chapter 9 and outline future work.

2 Preliminaries

In this chapter, we study preliminary concepts and principles that are relevant to this dissertation. Several main research domains have been covered: real-time data warehouse (DW), Extract-Transform-Load (ETL), streaming processing engines and scalable data stores.

We first start with an introduction to classical incremental view maintenance techniques used in traditional databases in Section 2.1 and then present how these techniques were further exploited in near real-time DWs in Section 2.2. Furthermore, we look into lazy view maintenance policy used in near real-time DWs in Section 2.3, which inspired us to design on-demand ETL for real-time analytics.

In Section 2.4, we compare several existing real-time DW solutions which also address real-time analytics. Furthermore, as it is common that most of these real-time DW systems utilize stream processing techniques for low latencies, we describe the architectures of several mainstream streaming processing engines in Section 2.5. In the end, we focus on the elasticity property addressed in the streaming systems at the engine level (in Section 2.6) and at the storage level (in Section 2.7).

2.1 Incremental View Maintenance

In the database research community, incremental view maintenance has been extensively studied and discussed at the end of the 20th century. One of the most representative work is [BLT86], in which two major contributions were made by Blakeley et al. First, they provided efficient detection of irrelevant updates to the views. Second, for relevant updates, they present a differential view update algorithm to refresh simple materialized views defined by Select-Projection-Join (SPJ) expressions. They defined *differential* forms of select, projection and join expressions, respectively, to identify which tuples must be inserted into or deleted from the SPJ views.

In [GM$^+$95], a complete survey on the incremental view maintenance techniques was provided. First, view maintenance problems were classified and studied along three dimensions: information (the data used to compute deltas), modification types (e.g., insertion, deletion) and langua-

ge (SPJ, aggregation, recursion, etc.) dimensions. Furthermore, several application domains were motivated by the incremental view maintenance techniques. One important application is *data warehousing* (DWH), since data warehouse tables can be seen as special materialized views. The same incremental maintenance techniques could be applied to ETL jobs to update DWH tables as well.

However, the view maintenance problem for warehouse views differs from the traditional one, since the view definition and the base data are decoupled in the DWH environment. Maintaining the consistency of warehouse data is challenging, since the data sources are autonomous and views of the data at the warehouse span multiple sources. Take a warehouse view $V = r_1 \bowtie r_2 \bowtie r_3$ which joins data from three separate sources r_1, r_2, and r_3 as an example. For an insertion $\Delta(r_1)$ on r_1, the warehouse view V can be refreshed by $\Delta(V) = \Delta(r_1) \bowtie r_2 \bowtie r_3$, which is calculated by a query Q joining the delta on r_1 with other two source data. The $\Delta(V)$ gets dirty when updates (e.g., deletion $\Delta(r_2)$) occur between the occurrence of $\Delta(r_1)$ and the execution of Q. The view maintenance anomaly can be avoided by either copying source data to the warehouse side (high resource and maintenance overhead) or issuing global transactions over data sources (infeasible due to source autonomy).

Zhuge et al., addressed the maintenace anomaly in [ZGMHW95, ZGMW96]. They developed the Eager Compensating Algorithm (ECA) and the Strobe family of algorithms to resolve anomalies. In the above example, the dirty $\Delta(V)$ can be compensated by $\Delta(r_2)$ at the warehouse before merging into the warehouse tables. Besides, they defined four levels of consistency for warehouse views: *convergence*, *weak consistency*, *strong consistency* and *completeness*. The Strobe family of algorithms guarantee strong consistency where each warehouse state reflects the same globally serializable schedule at each source without extra costs incurred by heavy global transactions.

2.2 Near Real-time Data Warehouses

Near real-time data warehouses address online DWH scenarios where warehouse tables are refreshed while OLAP queries are performed. Incremental view maintenance techniques are applied to the ETL jobs that take delta tuples derived from source tables and generate final deltas to update target warehouse tables incrementally. The batches of source deltas are shortened to windows of micro-sizes (e.g., several minutes), thus significantly reducing the execution latency and enabling online warehousing.

2.2.1 Incremental ETL

Jörg et al., made contributions to the domain of formalizing ETL jobs for incremental loading of data warehouses [JD08, JD09a, JD09b, BJ10]. The formalization is based on a common transformation operator model called Operator Hub Model (OHM) [DHW+08]. A set of equivalence-preserving transformation rules are introduced to derive incremental variants from initial ETL jobs in a platform-independent manner. Besides, the transformation rules also work for *partial* change data which is incurred by common Change-Data-Capture (CDC) processes. The generalization of the rules allows incremental view maintenance techniques to apply to various ETL development tools, e.g., ETL scripts, SQL transformation queries, or full-fledged business-intelligence (BI) products (e.g., IBM InfoSphere DataStage ETL [inf])

2.2.2 Near real-time ETL

In [VS09], Simitsis et al., gave a very detailed introduction to best practices of "Near Real-time ETL", which was previously referred to as *active data warehousing* [SVS05, KVP05]. It mainly focused on the general architecture of near real-time DWs and key components, e.g., CDC processes and Change Data Propagation (CDP) processes. The CDP processes execute incremental ETL jobs which propagate only the delta tuples derived from the source tables to the target DWs. Besides, the role of the data staging area is extended with extra functionality which relieves the overload of delta tuples due to high-rate ingestion from DWs and on the contrary, at a very high query rate, alleviates the burden of locking caused by concurrent read/write accesses to those shared data partitions in DWs. Moreover, two levels of parallelisms (i.e., pipeline parallelism and partition parallelism) are introduced that are frequently used to accelerate the executions of ETL jobs. Near real-time ETL enables business analysts to obtain instant reports derived from fresher DW data, as ETL jobs are executed more frequently in micro-batches (e.g., minutes) in an online fashion.

Furthermore, [DCSW09] claimed that the next-generation BI should be operational BI, where the data that is cleansed with high quality in the DWs should flow back to operational databases to increase business values. More general data flows are thereby attractive which combine the data transformation flows in ETL jobs with the data integration flows and analytical queries in DWs. In this way, global optimizations could be carried out from end to end in the entire flow according to different quality

metrics (performance, fault tolerance, freshness, etc.). Such general data flows are called *hybrid flows*.

Several research topics in global optimizations over hybrid flows have been addressed as follows. The work [SWDC10] addressed the challenge of identifying which operators in a hybrid flow are good candidates to materialize with the consideration of both fast recovery of ETL jobs and high freshness for warehouse data. Other research efforts are investigated in making decisions to deploy logical hybrid flows onto appropriate execution engines (relational databases, ETL engines or MapReduce) for various SLAs (e.g., performance, freshness) [SWCD12]. Moreover, [KVS13] studied several scheduling policies which cut a hybrid flow into multiple strata and schedule them to balance memory consumption and execution time.

Simiarly, Thiele et al., proposed the concept of Workload Balancing by Election (WINE) in [TFL09], which allows users to express their individual demands on the query response times (Quality of Service, QoS) and the freshness of request data (Quality of Data, QoD). Due to the fact that DW users' demands change over time and different users might have even conflicting demands, a scheduling algorithm is presented which balances the total $\sum QoS$ and $\sum QoD$ and allocates system resources to either queries or updates if any of them is prioritized.

2.3 Lazy Materialized View Maintenance

In near real-time data warehouses, warehouse views can be incrementally maintained using different refresh policies: *immediate*, *deferred* or *periodic* strategy. Deferred, *lazy* or *on-demand* view maintenance was first introduced by Hanson in [Han87] in the database domain.

Using immediate view maintenance, the update transaction that modifies base tables commits only after the materialized views are updated while deferred view maintenance is decoupled from the update transaction and refreshes the view data as queries arrive. Hanson provided a performance comparison among queries without materialized views, queries with materialized views using *immediate* and *deferred* view maintenance strategies based on the following metrics: the total fraction of operations that are updates (P), the selectivity factor of the view predicate (f), the fraction of the view retrieved by each query (f_v), and the number of tuples written by each update (l).

With experimental results, the following conclusions are reached. When P is high, f is high, or f_v is small, queries favor not using mate-

rialized views at all. In such case, it is more efficient to access base tables while materialized views are only effective if f_v is large. Higher values of P, f_v, and l favor deferred view maintenance over the immediate scheme. The immediate scheme has a slight advantage over deferred maintenance only if P is low.

Deferred maintenance has an advantage over the immediate scheme that the buffered updates could be used to refresh the materialized views asynchronously whenever there is idle CPU and disk time available, which reduces the latencies of both update transactions and queries.

This thinking was exploited by Zhou et al., in [ZLE07] and systematically implemented in Microsoft SQL Server [mic]. Actual view maintenance tasks are done by low-priority background jobs when the system has free CPU cycles. Not only are the update transactions free from view maintenance, but also most of the maintenance cost are hidden from users' queries. Furthermore, they exploited row versioning to ensure the queries to see consistent snapshots of the database while not blocking maintenance jobs. Moreover, the efficiency of view maintenance is improved by merging maintenance jobs and eliminating redundant updates in delta streams.

2.3.1 Right-time ETL

As a counterpart to the materialized views, lazy view maintenance can also be adapted to the ETL domain. In [TPL08], Thomsen et al., described their proposal of "Right-Time ETL (RiTE)". A middleware system was implemented which provides DW data with INSERT-like data availability, but at bulk-load speeds.

RiTE only addresses the loading phase of ETL processes. As shown in Figure 2.1, the outputs of ETL jobs are generated by the ETL processes (*producer*) and are temporarily buffered in the ETL staging area until they are queried by *consumers* or there are free CPU cycles for DW systems to ingest. At the DW side, a main-memory based *catalyst table* is introduced to ingest the outputs from ETL and to periodically merge data into disk files. A query issued to the DW is executed over a *union* view of the catalyst table and the on-disk tables.

All of the outputs of ETL jobs are indexed with transaction commit timestamps. In addition, DW users can demand specific data freshness of their data analysis by setting data currencies in SQL queries. RiTE is able to tell whether the current DW-side data is complete enough to answer the query by comparing the expected query accuracy with the maximum

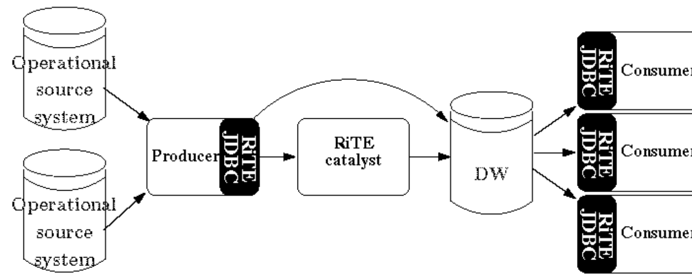


Figure 2.1: Architecture for Right-time ETL [TPL08]

timestamp of rows buffered in the catalyst table. If not, RiTE forces the ETL-side producer to flush required data to the DW-side catalyst table for query processing.

A key problem is that RiTE heavily relies on the main-memory catalyst table, which needs to be further extended for *Big Data* scenarios. Besides, in order to guarantee end-to-end data consistency for real-time analytics, crucial efforts should also be investigated in the extraction and transformation phases of ETL jobs in addition to the loading phase.

2.3.2 Relaxed Currency in SQL

Using lazy maintenance, a warehouse view is refreshed either when the system has free cycles or when a query references the view. The source-side update transactions no longer bear the maintenance costs which in turn, are amortized by OLAP queries. The maintenance cost leads to longer response times of OLAP queries, which can still be further shortened if the expected currency of view data can be relaxed. With relaxed currency given in a query, warehouse systems are able to tell whether the corresponding view is fresh enough to answer this query. If so, the maintenance task can be postponed to a later point in time when there is idle CPU, thus offering the incoming query a free ride.

Therefore, it is crucial for users to specify their freshness needs in their OLAP queries. Guo et al., proposed a *currency* clause **CURRENCY BOUND** for SQL syntax in their mid-tier database cache prototype [GLRG04, GLR05]. The database cache system allows each SQL client to maintain a local cache buffering materialized views, which are transactional replicas of database state stored in a remote back-end server. The local replicas are updated in an asynchronous manner, thus not always up to date. With the currency clause, users can specify how stale the accessed data is allowed at most. Therefore, the SQL client is able to check whether

the local cache is capable of answering a user query, otherwise, a compensation query is issued to the remote back-end server to fetch missing data to meet the currency demand of the user query. For example, **SELECT * FROM authors CURRENCY BOUND 10 MIN** indicates that the rows from the **authors** views in the local cache must be at most 10 minutes out of date.

As the local cache and the back-end server are similar to the warehouse views and the source tables in the DWH environment, we believe that it is worth to support the currency clause for OLAP queries to accelerate query processing and alleviate on-demand view maintenance overhead.

2.4 Real-time Data Warehouse

As defined in [ora14], real-time data warehouses demand that source changes are captured and immediately propagated to DWs within seconds. The existing real-time data warehouse solutions are mainly guided by two principles: *one size fits all* and *one size does not fit all*. Examples of one-size-fits-all databases are Hyper [KN11] and SAP HANA [FCP+12], which utilize modern hardware to manage OLTP & OLAP data and workloads in single systems. On the other hand, BigDAWG [EDS+15, MZT+16] is an example following the one-size-does-not-fit-all principle [SC05], which unifies domain-specific engines as a single, logical unit and utilizes streaming processing techniques to speed up ETL jobs among them, thus reducing the time window of data ingestion and data synchronization to some extent.

2.4.1 One-size-fits-all Databases

Hyper and SAP HANA are in-memory databases where data is stored entirely in memory and in a columnar manner, which guarantees instant query processing. In Hyper, updates in OLTP workloads are executed sequentially in a single OLTP process while each OLAP query session is always executed on a *snapshot* of database state in a child process *forked* from the parent OLTP process. In SAP HANA engines [FCP+12, GPA+15], in memory data is organized in log-structured merge tree format. Continuously incoming updates are first appended to a differential buffer and later merged into the main store for efficient query processing.

As another example, IBM Data Analytics Accelerator (IDAA) [BBD+15] serves as an OLAP engine originated from the Blink system [RSQ+08]. It enhances IBM DB2 for z/OS with efficient BI analytics. OLTP data is

offloaded from DB2 to IDAA and stored entirely in memory in a columnar format. For real-time reporting or predictive model training, data can be scanned significantly fast using SIMD instructions without indices.

2.4.2 Polystore & Streaming ETL

Stonebraker et al., have argued that the "one size fits all" concept is no longer applicable to the database market while the complexity of newly-emerging applications could only be overcome by domain-specific engines [SC05]. A new concept called *polystore* emerged which serves a spectrum of applications through a common front-end parser.

BigDAWG is a polystore system for unifying multiple storage engines, which resembles federated database systems like Garlic [CHS+95]. It was designed based on the *one-size-does-not-fit-all* theory. As each storage engine is optimized for its specific workload, BigDAWG is capable of handling a variety of complex analytics. For example, an application of anomaly detection for heart waveforms requires the underlying engines to process both real-time ingestion and analytics of historical data. In BigDAWG, such an application request can be split into sub-queries for a streaming engine and a MapReduce-based platform and executed in a federated manner.

In order to efficiently migrate data across different storage engines at runtime, Meehan et al., [MAZ+17] proposed streaming ETL to address lower execution latency for data ingestion and data transformation. Streaming ETL is provided by a transactional stream processing system called S-Store [MTZ+15], which guarantees ordered execution, exactly-once processing and ACID properties when accessing shared mutable states. Due to the diversity of analytics, the same data set might be processed in different engines for efficiency. With low-latency streaming ETL support, data could be efficiently migrated across processing engines wherever it is needed. Furthermore, a dynamic, workload-driven approach is proposed for caching and data placement in polystores [DMTZ17], which provides strict performance guarantees for analytical queries.

2.4.3 Stream Warehouse

Golab et al., [GJSS09, GJS09, GJS11] mainly focused on stream DWs, which combine the features of traditional DWs and data stream systems and mainly serve applications like online stock trading, credit card fraud detection and network monitoring. In a stream DW, data is divided into

multiple partitions as consecutive time windows, while each query accesses data from the latest timestamp preceding its submission time in each partition. Besides, ETL jobs are triggered immediately, as new data arrive from source data streams. Based on the *data staleness* metric defined for target warehouse tables, ETL jobs are scheduled for minimum average staleness under resource control.

In [GJ11], they proposed their *temporal consistency* which is used to quantify and monitor the data quality in a real-time stream warehouse. For each data partition of a certain time window, three levels of query consistency are defined, i.e., *open*, *closed* and *complete*.

A partition is marked as open for a query if data exist or might exist in it. A closed partition implies that the final data range has been fixed, even though all the data haven't arrived completely and are expected to be complete in a limited time window, which means query results could be incomplete. The complete level is the strongest query consistency level. A complete partition is closed and all the data have arrived.

2.5 Streaming Processing Engines

Real-time DWs rely on stream processing techniques to continuously refresh warehouse tables and to enable real-time analytics. The streaming processing engines gradually evolved over several generations and now have been used in the ETL domain.

2.5.1 Aurora & Borealis

Aurora [ACÇ+03, CÇR+03] is a first-generation stream processing engine, which addresses the performance and processing requirements of stream-based applications. Applications settle inside Aurora in the form of *continous queries*, each of which continuously processes unbounded data streams and produces results back to the front-ends. Quality-of-Services (QoS) (e.g., latency) can be specified in Aurora's queries and guaranteed during query executions.

Figure 2.2 illustrates the architecture of the Aurora runtime engine, which is fundamentally a data-flow system. Each continuous query is represented as a directed acyclic graph (DAG) of a set of *operators* (called *boxes* in Aurora) and *tuple queues* while each operator box consists of one or more input queues and output queues. All the tuple queues are stored in a persistent storage and maintained by a storage manager, which manages the queue buffer for boxes to access their queues.

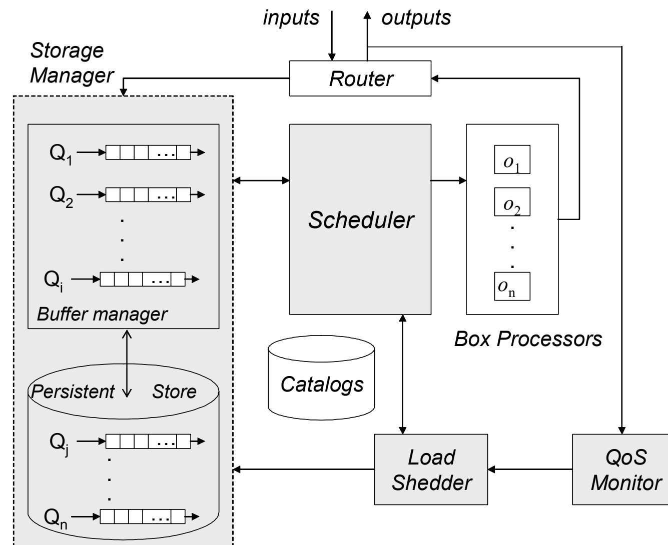


Figure 2.2: Aurora Runtime Engine [CÇR+03]

The key component of Aurora is the *scheduler* which controls the box executions at runtime. The scheduler continuously generates stream tasks which define *which boxes to execute next* and *how many tuples to process*. The decisions are made by estimating the box processing costs (i.e., tuples per time) based on several performance metrics (e.g., selectivity, execution times, thread context switches and memory utilization). The stream tasks are executed by a multi-threaded *box processor* and their outputs are routed to the target applications or back to the tuple queues for subsequent boxes to process. Furthermore, a *QoS monitor* keeps monitoring the outputs of boxes and checks whether the QoS are met or not. In case of overload situation, stream tasks can be shed by a *load schedder* until the QoS reach an acceptable level.

Borealis [AAB+05] is a second-generation distributed stream processing engine that inherits core processing functionality from Aurora. In addition to the performance requirement, Borealis addresses scalable, highly distributed resource allocation, optimization capabilities and fault tolerance for applications that run in a distributed environment. The DAG of a continuous query is split into fragments and executed in multiple nodes that exchange tuples across network.

The architecture of Borealis is shown in Figure 2.3. Borealis nodes exchange meta-data and control messages through *transport independent RPC* while transferring tuples through *IO queues*. With an issued query,

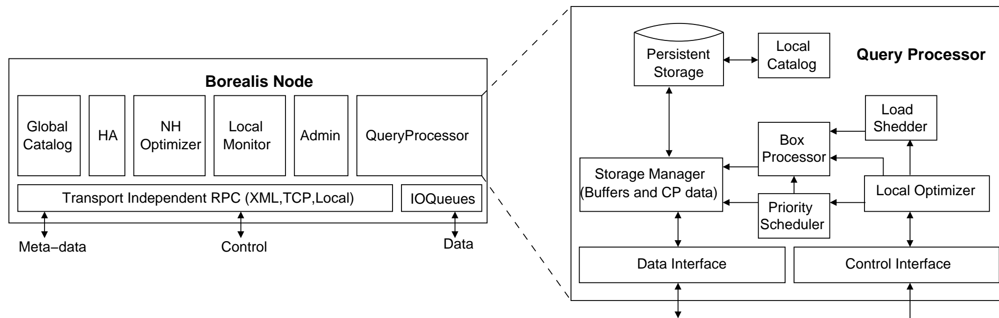


Figure 2.3: Borealis Architecture [AAB+05]

the *admin* module cuts the compiled DAG to fragments, moves them to remote nodes and possibly, leaves one fragment for the local *query processor* to execute. Most of the components in the query processor are similar to those in Aurora while an additional *local optimizer* serves as a proxy to forward local-processing messages from the admin module to the query processor. Besides, IO queues are used to transfer the outputs of local boxes to remote ones.

A *local monitor* collects local load information and broadcasts it to remote *neighbourhood optimizers*. In this way, all the neighbourhood optimizers share global load information and possibly in a “busy” node, one local neighbourhood optimizer can issue its local admin module to move some running fragments to “idle” nodes for load balance.

Furthermore, a *global catalog* holds the complete DAG fragments with corresponding node locations. Hence, once the *high availability* module detects a “dead” node in case of node failure, it can ask its local admin to take over the processing of fragments from that dead node.

2.5.2 Apache Storm

As stated in [TTS+14], Storm [sto] is a scalable, resilient, and extensible real-time streaming processing framework with at least once processing semantics. Storm has been widely used at Twitter to process large-scale computations on streaming data (e.g., tweets) in real-time, since it can be easily scaled out if the resources are not enough to meet the application needs and also importantly, it is fault-tolerant in case of node failures in large clusters.

Users’ stream applications are directed graphs called *topologies* in Storm. The topology nodes are operators which are classified as sources (called *spouts*) and transformations (called *bolts*). Each spout or bolt is a lo-

gical operator and consists of a set of tasks, each of which processes a part of the outputs generated from its upstream operator. An upstream spout or bolt task is connected via a tuple queue with its downstream task.

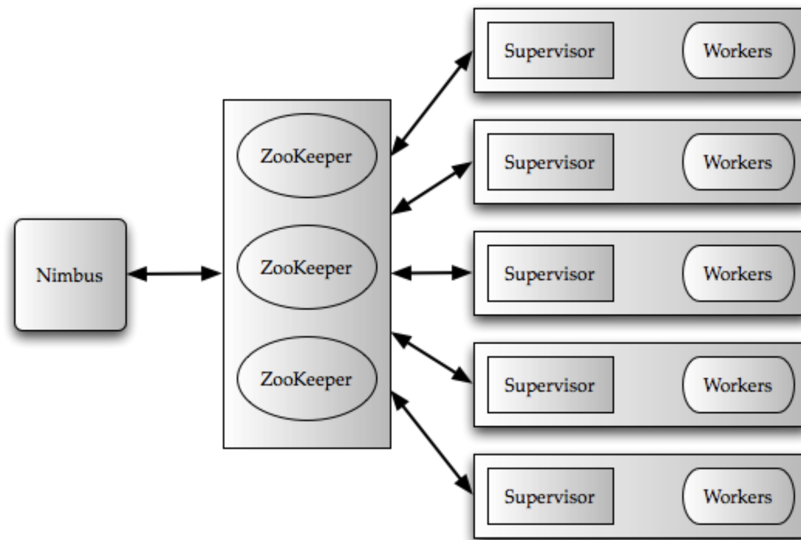


Figure 2.4: High Level Architecture of Storm [TTS+14]

As shown in Figure 2.4, the architecture of Storm follows a master-slave structure. A node (called *Nimbus*) acts as the master node which is responsible for distributing a user-submitted topology to the slave nodes (called *Supervisors*). All Supervisor nodes report their aliveness and available resources to Nimbus via Zookeeper while each Supervisor node runs one or more *worker processes* and each worker further runs one or more *executor threads*. All the tasks of a certain spout or bolt are assigned to the executors and processed continuously on streaming data across Supervisor nodes in a Storm cluster. More specifically, one dedicated receiver thread inside a worker process listens on a TCP/IP port and forwards each incoming tuple to the input queue of its destination executor based on an internal *task id*. The executor threads processes the corresponding task and buffers the outputs in its output queue, which are further collected by a global transfer queue. Finally, another dedicated sender thread sends all outgoing tuples to downstream worker processes.

Storm guarantees at least once processing semantics, meaning that each tuple will be processed at least once in a topology. To achieve this semantics, an *acker* bolt is added to a topology and tracks the execution status of each spout or bolt for every tuple based on a *tuple id* maintained by the receiving spout. Once a tuple is successfully processed in a topology, the

acker bolt notifies the spout of this event and the tuple id can be released. Otherwise, the spout needs to replay this tuple in the next iteration.

For applications that require exactly-once processing semantics, Trident [tri] is provided which is a high-level abstraction on top of Storm. In Trident, unbounded input streams are split into batches which are attached with *transaction ids* and executed in sequence when accessing external states. By storing and comparing the transaction id of a retried batch with external state, the updates of this batch are idempotent in the face of failures.

2.5.3 Apache Spark Streaming

Spark Streaming [ZDL+13] is a distributed stream processing framework built on top of Spark [spa] that is a highly efficient batch processing engine. In addition to reusing the existing features like parallel operators, *resilient distributed datasets* (RDDs) [ZCD+12], etc. from Spark, a new programming model called *discretized streams* (D-Streams) is proposed, which offers a high-level functional programming API, strong consistency, and efficient fault recovery. The highlight of D-Streams is to treat a streaming computation as *a series of deterministic batch computations on small time intervals*.

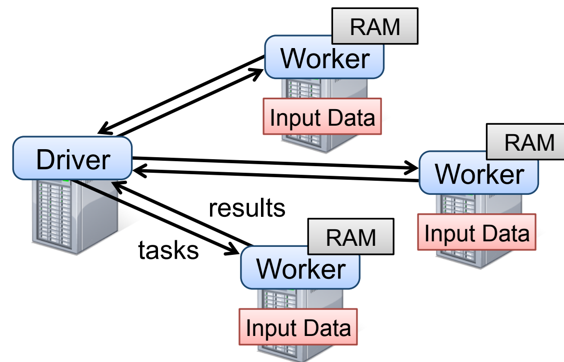


Figure 2.5: Spark Runtime [ZCD+12]

Spark supports tasks as small as 100ms and job latencies as low as a second, since traditional batch systems (e.g., Apache Hadoop [Whi12]) store intermediate state on disk between stages of jobs while Spark keeps them in memory as RDDs. RDD is a storage abstraction that tracks the *lineage* (the set of deterministic operations used to build in-memory data), thus avoiding expensive data replication for fault-tolerance. When a node fails, Spark simply recomputes the lost RDD paritions by rerunning necessary

operations (specified in the RDD) on the data that is still available in the cluster. Such simple recovery is similar to batch data-flow systems. Moreover, as these recovery operations run in parallel, Spark provides faster recovery than upstream backup, where upstream nodes buffer state and replay them to a standby node for a failed downstream node.

Figure 2.5 illustrates the execution runtime of a Spark cluster. A Spark application is a DAG consisting of deterministic parallel operations, such as *map*, *join* and *groupBy*. An application is initially submitted to a *driver* process where the DAG is cut into *stages* at the boundaries of *shuffle/repartition* operations. A stage is fundamentally a *taskSet*, i.e., a set of *tasks*, each of which executes pipelined operations on a RDD partition. Meanwhile, the driver applies for resources to start multiple *executor processes* on *worker nodes* in the cluster. The executor processes periodically send heartbeats to the driver to update their available-resource information, based on which the driver schedules taskSets to specific executors for processing.

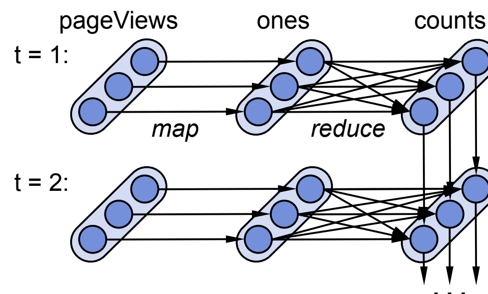


Figure 2.6: Lineage Graph for D-Streams and RDDs [ZDL+13]

A Spark Streaming program is a DAG of D-Streams, each of which groups together a series of RDDs. Take a running count of page view events by URL as an example (see Figure 2.6). A D-stream called *pageViews* consists of a series of RDDs and each *pageViews* RDD groups incoming events during each 1-second interval. Each *pageViews* RDD is then transformed to a RDD of (URL, 1) pairs and further used to perform a running count by a *reduce* RDD, which belong to a D-Stream *ones* and a D-Stream *counts*, respectively.

Spark Streaming guarantees at least once processing semantics by periodically checkpointing state RDDs. For example, in case of node failure on the sixth RDD, the state can be recovered by replaying the events from the latest checkpoint (e.g., the fifth RDD). Similar to Trident, exactly-once processing semantics can be achieved by letting users to enhance business

logic with additional transactional properties. For example, a *transaction id* can be derived from both a RDD partition id and the time interval, and further used to commit the data once the processing on that partition is finished, thus skipping recomputing the same partition upon recovery.

2.5.4 Apache Flink

As defined in [fli], Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. In contrast to Spark, Flink's core is a streaming dataflow engine which unifies different computation models (real-time analysis, continuous streams and batch processing) together. Flink utilizes the *window* mechanism in stream processing to cover both stream and batch cases, i.e., batch programs are special cases of streaming programs, where all records implicitly belong to one all-encompassing window [CKE+15].

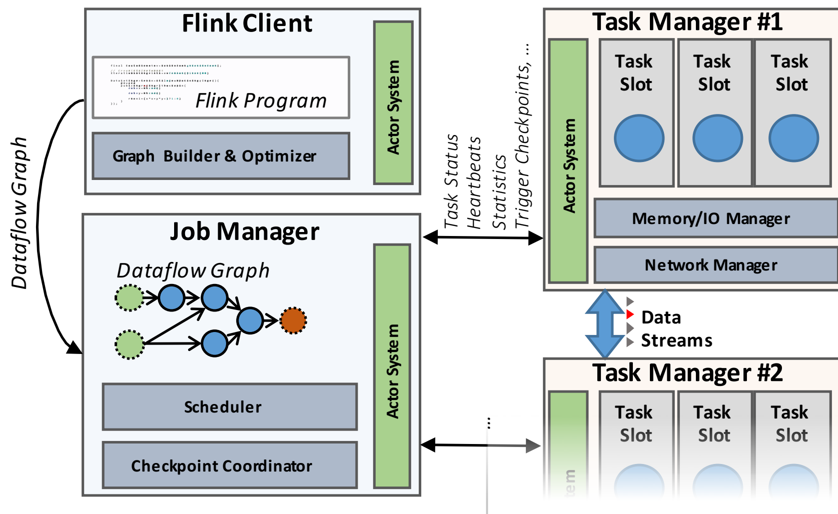


Figure 2.7: Flink Process Model [CKE+15]

Flink programs are internally represented as *dataflow graphs*, which are DAGs with stateful operators as nodes and data streams as edges. A stateful operator consists of one or more *tasks* and a data stream is split into one or more *stream partitions* for tasks of successive operators to exchange tuples. The process model of Flink is shown in Figure 2.7. A *client* process transforms a Flink program to a dataflow graph and submits it to a *Job Manager* process. The job manager process is responsible for scheduling operator tasks to distributed *Task Manager* processes, coordinating

periodic checkpoints, and performing load-balancing, health-check and failure recovery. The tasks of a stateful operator can be executed across one or more multi-threaded task manager processes and the producing data stream buffers are sent to the tasks of consumer operators through network connections. Flink uses intermediate data streams to exchange data between operators. In Figure 2.8, two types of data streams: *pipelined* and *blocking* intermediate streams are shown. Every incoming tuple in a pipelined stream can be immediately fetched by a consumer task to process without materializing it in memory while a blocking stream buffers relevant tuples for consumption until a certain condition fires, e.g., sort-merge joins.

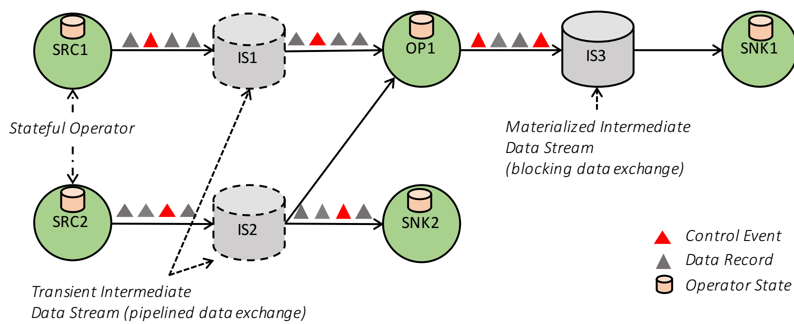


Figure 2.8: Flink Dataflow Graph Example [CKE+15]

Flink offers at-least-once processing semantics by checkpointing and replaying messages from durable source message queues. The checkpoint mechanism is called *asynchronous barrier snapshotting* [CFE+15], which is different from traditional mechanisms. Specific control tuples (called *barriers*) are inserted into the input streams together with other data tuples. Each time when an operator task meets a barrier from the input stream, it flushes its in-memory state to durable storage and forwards the barrier downstream. In this way, when the job manager process collects the barriers from all the sink tasks, a global snapshot of the dataflow graph is completed. Upon recovery, all the tasks revert the state back to the latest global checkpoint and replay the messages from the source offsets stored in the checkpoint. To achieve exactly-once processing semantics, duplicate tuples can be detected by downstream tasks via the *sequence numbers* augmented from the sources.

2.5.5 S-store

S-Store [MTZ+15, TZM+15] is a stream-oriented extension to a traditional OLTP system called H-Store [KKN+08] for processing stream transactions with well-defined correctness guarantees. It addresses the inconsistencies of streaming applications due to lack of isolation from ACID transactions, when multiple stream operators access and update *shared mutable state* in streaming systems. For streaming workloads that require transactional state, S-Store outperforms pure OLTP systems and pure streaming systems in terms of throughput while it guarantees the properties of both OLTP and streaming systems: ACID, ordered execution, and exactly-once processing.

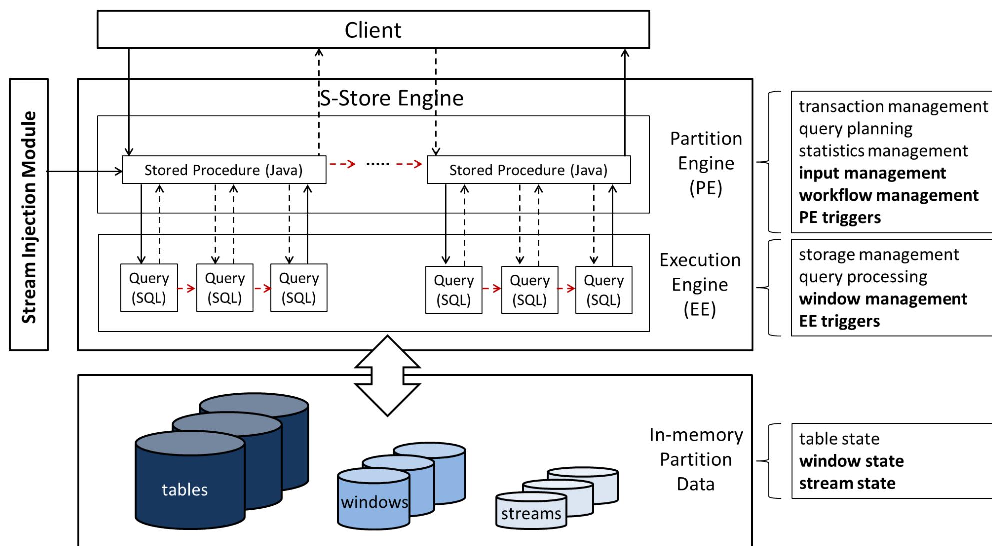


Figure 2.9: S-Store Architecture [MTZ+15]

A streaming application is a DAG with nodes as stored procedures and edges as execution ordering. A stream is a series of batches of tuples that are augmented with source timestamps or *batch ids*. Tuples with the same batch-id are logically grouped as an *atomic batch*. With arrival of an atomic batch, the corresponding stored procedure is triggered to process the tuples of this batch as a unit in a so-called *streaming transaction*. The output tuples of a streaming transaction share the same batch id of the input atomic batch and become the atomic batch for downstream transaction. For a stored procedure S_i, we denote a streaming transaction T over an atomic batch b_j as $T_{i,j}(b_j)$. An edge $S_1 \rightarrow S_2$ in a DAG means that S_1 precedes S_2 ($S_1 \prec S_2$) and the following order execution rules must be guaranteed: *da-*

taflow graph order constraint: $T_{1,r}(b_r) \prec T_{2,r}(b_r)$ and *stream order constraint:* $T_{1,1}(b_1) \prec T_{1,2}(b_2) \prec \ldots T_{1,r}(b_r)$.

Figure 2.9 depicts the architecture of S-Store. S-Store offers additional streaming functionality to H-Store, which is a distributed, in-memory database. Transactions are executed in a distributed manner by master processes (*Partition Engine* PE) and slave processes (*Execution Engine* EE). In S-Store, streams are in-memory H-Store tables with one timestamp column which distinguishes the batches of tuples. When an upstream transaction commits on an atomic batch in a PE, the tuples of this batch become visible in the output stream table, which activates a transaction execution of the downstream stored procedure in another PE by a database trigger. Triggers (red arrows in Figure 2.9) are used to enable push-based, data-driven streaming processing over H-Store. Furthermore, S-Store naturally guarantees exactly-once processing semantics by skipping recomputing committed streaming transactions on failed batches.

2.5.6 Pentaho Kettle

Pentaho Data Integration (PDI) - Kettle [CBVD10] is an open-source ETL tool that is widely used in the research community and provides a full-fledged set of transformation operations (called *steps*). Kettle jobs process data in a streaming manner, i.e., batches of files are taken as inputs and processed row by row in each step. During the flow execution, each step is running as an individual thread. Step threads are connected with each other through an in-memory queue called *RowSet*. The results of a preceding step are put in an output rowset which in turn serves as the input rowset of its downstream step.

Kettle jobs can also be executed in a cluster mode in which multiple copies of the same flow instance run in parallel across distributed cluster nodes. The cluster mode follows a master/slave architecture. The output rows of an upstream subflow are partitioned based on a specific partition schema in the master node and further distributed to target slave nodes for further processing of the downstream subflow.

2.6 Elasticity in Streaming Engines

Elasticity is an appealing feature which has been addressed in most of the full-fledged stream processing engines. It enables the data flows to scale out/in automatically under varying load at runtime without manual

intervention, which should also be considered when designing our on-demand (streaming) ETL solution.

Schneider et al., [SAG+09] described how to make stateless operators adaptive to changing workload by dynamically balancing load across working threads through a global tuple queue. A control algorithm was designed to identify the peak rate (maximum processing throughput) and to decide on correct number of parallel working threads at runtime. They improved this algorithm in [GSHW13] by additionally considering the tuple block time in the buffers between operators. Besides, they addressed the state migration problem for partitioned stateful operators through a state management API. When a scale out is triggered, local state is re-partitioned to multiple packages according to new partition schema and shuffled across new partitioned operators through a backend database. During the state migration, the upstream operators are blocked to send output tuples, thus resulting in back pressure.

Fernandez et al., [CFMKP13] addressed the state migration problem by backing up the processing state of a bottleneck operator periodically in its upstream operator and repartitioned for scale out. Buffers are used in upstream operators to temporarily ensure consistent state during scale out, which might trigger a scale out of the upstream operator if the data rates still rise. In contrast, Wu et al., [WT15] pre-partition future operator state at deployment time and distribute state partition replicas across cluster nodes at runtime. Transparent workload migration is achieved by spawning new threads on remote nodes where backup partition replicas are located, releasing original ones and re-routing outputs from upstream operators. However, skewed workload may result in hotspot partitions, thus requiring additional re-partitioning logic to split skewed data under load at runtime.

Other work exists which covers elastic queue or buffer data structures in data flow systems. Karakasidis et al., [KVP05] proposed a framework for the implementation of active data warehousing. ETL activities are considered as queues while an ETL flow is transformed to a queue network, where queue theory can be performed to predict the mean delay and the queue length of ETL queues, given the source production rates and the processing power of the staging area. However, they did not address the scalability of queues, although the idea of pipelining blocks of tuples in ETL queues was already there. Furthermore, as introduced in [CKE+15], Apache Flink categorizes the intermediate data streams as *pipelined* streams and *blocking* streams. For pipelined streams, intermediate buffer pools are used to compensate in case of short-term throughput fluctuations.

2.7 Scalable NoSQL Data Store: HBase

HBase [Geo11] is an open source, wide-column NoSQL database [Cat11] modeled after Google's BigTable [CDG+08]. It runs on top of HDFS (Hadoop Distributed File System), which is a component of Apache Hadoop [Whi12] (open-source implementation of Google's MapReduce [DG08]). It delivers real-time read/write access to data stored in HDFS. As similar to other wide-column NoSQL databases (e.g. Cassandra [LM10]), HBase is targeted at online Web and mobile applications as well as data warehousing applications, since it is a key component of the Hadoop family and it stores data directly in the underlying HDFS. Generally, long-running, batch-oriented analytics can be executed over data in HDFS through Hive [TSJ+09] by running MapReduce jobs, while near real-time analytical queries can be issued to HBase for quick range scans [CST+10] over the same data in HDFS.

2.7.1 System Design

Internally, HBase stores a table as a scalable, distributed, sorted, multidimensional *map* where each value is indexed by a key concatenated with a user-defined row key, a column family name, a column name and a timestamp. This map is horizontally partitioned into a set of data partitions (called *Regions*) with non-overlapping key ranges. HBase follows a master/slave architecture where all the regions are stored only in the slave nodes (called *Region Server*) while the master node has only meta-data information about server locations of regions. Based on the meta-data information of regions, data manipulations on a table are re-directed to remote region servers where actual read and write operations are carried out.

Figure 2.10 illustrates the internal structure of HBase regions. The design of region follows log-structured merge (LSM) tree. When adding a tuple (a key-value pair) to a region, this tuple is first buffered in an in-memory data structure called *MemStore*. The MemStore has a limited size and is implemented as a skip list where tuples can be searched, inserted or deleted in a logarithmic time. Once a MemStore fills up, all its tuples are first sorted in a lexicographical order and then flushed onto disk as a new *StoreFile*. Therefore, a region is composed of an in-memory MemStore and multiple on-disk StoreFiles. To lookup one or more tuples in a region, the region server performs *scans* which internally open scanners on the MemStore and the StoreFiles to identify the target tuples.

In a single region, Multi-Version Concurrency Control (MVCC) is adopted to isolate read and write operations. For each write operation, a

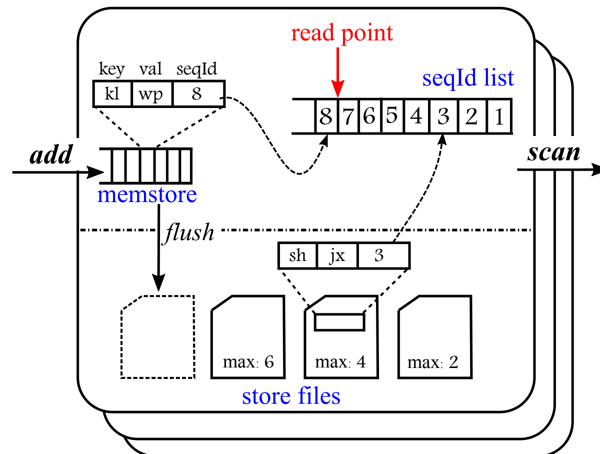


Figure 2.10: HBase Region Structure

sequence id is assigned and attached to all its tuples. Furthermore, concurrent write operations are only allowed to commit in sequence based on their assigned sequence ids. The tuples of uncommitted writes are invisible to scans (with READ COMMITTED as the isolation level), even though they have been essentially buffered in the MemStore. This is achieved by assigning a global variable (called *read point*) value to all the scans and filtering out uncommitted tuples based on their local read points. A global read point always points to the greatest sequence id from the committed writes in a region at any time, while each scan has a local read point whose value is initialized with the global read point value at the time when the scan arrives. The scan carries the local read point throughout its entire life cycle and only access tuples with sequence ids smaller than its local read point.

In Figure 2.10, a StoreFile scanner of a scan request has its local read point as 7, as 7 was the greatest sequence id at the time when this scan request came. A tuple with a key-value pair (key:"sh"; val:"jx"; seqId: 3) is thereby returned by this scanner, since its sequence id 3 is smaller than the read point. However, a tuple with a key-value pair (key:"kl"; val:"wp"; seqId: 8) in the MemStore would never be reached by the MemStore scanner of this scan, since the local read point precedes the sequence id 8.

For a scan spanning across multiple regions, the read points could be assigned to its sub-requests on specific regions at different time, thus leading to inconsistency from users' perspective. Therefore, strong consistency is only guaranteed at the region level.

2.7.2 Compaction

The reason why HBase flushes MemStores to new StoreFiles, instead of merging them into existing ones, is that HDFS does not support updating files due to data replication and furthermore, flushing is efficient using sequential writes. However, as more and more small StoreFiles are generated, the scan performance falls since more files need to be opened and scanned. Therefore, several improvements have been introduced to accelerate the performance of searching row keys in StoreFiles.

First, a StoreFile can be augmented with a *bloom filter* which checks whether a given row key is *not* contained in this StoreFile. Hence, the StoreFile scanners are able to skip some StoreFiles even though the rest of the StoreFiles might be false-positive. Second, if the bloom filter cannot guarantee the absence of the row key for a StoreFile, the search continues and uses a *block index* of a StoreFile to efficiently locate the first block to start. As tuples in a block were sorted before being flushed to the disk, the time complexity to address the target tuples is O(log(n)).

Unfortunately, since there is no key-lookup index maintained for the StoreFiles in a region, all the StoreFiles have to be searched and the search complexity is proportional to the number of the StoreFiles, i.e., O(mlogn). Therefore, HBase uses periodic *compaction* to mitigate the overhead of scans by merging small StoreFiles to larger ones. The compaction process scans two or more StoreFiles and creates a new single StoreFile, thus improving the scan performance as the number of StoreFiles decreases. However, the drawback is that the regions become unavailable as the read and write operations are forbidden during the period of compaction.

2.7.3 Split & Load Balancing

HBase achieves its scalability by auto-splitting and load balancing. If a region is overloaded under current workload, HBase *splits* this heavy region into two and migrates these two daughter regions to remote region servers for load balancing.

More specifically, as the compaction process always generates larger StoreFiles, auto-splitting is triggered when the size of the largest StoreFile in a region exceeds a configurable threshold. The *split point* is selected as the median row key from the tuples in the largest StoreFile. A consecutive key range of the splitting region is cut into two on this split point. To migrate new daughter regions to remote region servers, only links to the blocks of old StoreFiles are moved instead of copying StoreFiles across HDFS nodes. Once new regions are migrated, the meta-data information

is updated in the master node. These links are kept until next compaction starts, which writes new StoreFiles.

Normally load balancing is triggered periodically or directly after auto-splitting in HBase. Regions are moved from overloaded region servers to idle region servers. By default, the load on a region server is considered as the number of regions instead of data locality and storage overhead. Once a region with good data locality is moved, data locality decreases which incurs high network overhead for scans. Hence, an improved load balancer (called StochasticLoadBalancer) is introduced which takes data locality and region size into account when redistributing regions.

2.7.4 Bulk Loading

Traditional databases (e.g., MySQL [mys]) use the B-tree as an index structure for fast read and write on large tables. One crucial step of generic bulk loading in traditional databases is an index-construction process. Using a classical sort-based bulk loading approach, the entire data set is pre-sorted (O(nlogn)) and grouped in file blocks as index leaf nodes. A B-tree index can be easily built from this set of sorted leaf nodes in a bottom-up fashion from scratch. In contrast, inserting the tuples from the same data sets once at a time in a top-down fashion without pre-sorting incurs overhead, i.e., a lot of splits on index internal nodes and a large number of disk seeks with random I/O.

Similarly, as HBase is built directly on HDFS, an efficient MapReduce-based bulk loading approach is used to directly transform HDFS files into HBase StoreFiles, which skips the normal write path in HBase. The map tasks transform each text line to a Put object (an HBase-specific insert object) and send it to a specific region server while the reduce tasks sort these Put objects and generate final HFiles. This process is much faster than normal HBase writes as it exploits batch processing on HDFS. In addition, each row contains only one version throughout all StoreFiles, which accelerates future scans with the help of bloom filters set in the StoreFiles.

3 Incremental ETL Pipeline

As detailed in Chapter 1, our main goal is to make ETL processing runtime be aware of the real freshness needs derived from users' input OLAP queries, i.e., to pay the ETL costs just as needed and make ETL *on-demand*.

Based on the working scope illustrated in Figure 1.1, we briefly explained several primary challenges that need to be addressed to achieve on-demand ETL. In this chapter, we first start with the definition of a consistency model we introduced to address data consistency in the global picture, and later focus on our incremental ETL pipeline solution which achieves this objective.

The remainder of this chapter is organized as follows. We give an introduction to the computational model in Section 3.1, based on which, we present our incremental ETL pipeline in Section 3.2. A formal definition of our consistency model is given in Section 3.3 and the workload scheduler solution for achieving this consistency model is described in Section 3.4. We further address consistency issues in ETL pipelines in Section 3.5 and present experimental results in Section 3.6.

3.1 The Computational Model

In this section, we describe the computational model for our on-demand ETL. We use a dataflow system to propagate source deltas to the data warehouse and the ETL transformation programs are interpreted as *dataflow graphs*. As shown in Figure 3.1, a dataflow graph is a directed acyclic graph $G(V, E)$, in which nodes $v \in V$ represent ETL transformation operators or user-defined procedures (in triangle form), and edges $e \in E$ are *delta streams* used to transfer deltas from provider operators to consumer operators. A delta stream is an ordered, unbounded collection of delta tuples (Δ: insertions (I), deletions (D) and updates (U)) and it can be implemented as an in-memory queue, a database table or a file. There are two types of delta streams: *source delta streams* (e.g. streams for ΔR and ΔS) and *interior delta streams*. The source delta streams buffer source delta tuples that are captured by an independent Change-Data-Capture (CDC) process and maintained in commit timestamp order in terms of source-

local transactions. The interior delta stream stores the output deltas that are processed by the provider operator and at the same time, transfers them to the consumer operator. Hence, the same delta stream can be either the input or the output delta stream for two different, consecutive operators.

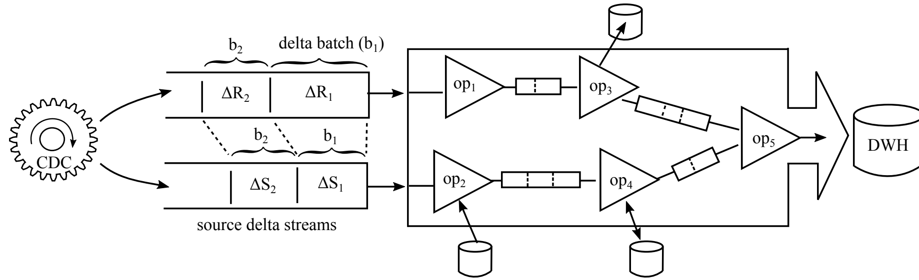


Figure 3.1: Dataflow Graph

Moreover, an event of a query arrival at timestamp t_i groups all source deltas with commit-time(Δ)< t_i in each source delta stream into a *delta batch* b_i and constructs a *maintenance job* m_i. Each delta batch b_i is a finite, contiguous subsequence of a delta stream and each tuple in b_i contains not only general information for incremental processing (e.g. change flag (I, D, U), change sequence number), but also the id of the maintenance job m_i. All the tuples in b_i have the same maintenance job id and should be processed together as a unit in subsequent transformation operators (e.g. op_1 and op_2). The output tuples after a delta batch processing are also assigned the same maintenance job id and are grouped into a new delta batch for downstream processing (e.g. op_3 and op_4). The maintenance job m_i is an abstraction of one maintenance flow execution where all the operators in the dataflow graph process the delta batches referring to the same job in their owning delta streams. (We use the terms "delta batch" and "maintenance job" interchangeably to refer to the delta tuples used in one run of each transformation operator.)

With a sequence of incoming queries, the source delta streams are split to contiguous, non-overlapping delta batches. A list of chained maintenance jobs are created for the dataflow graph to process. To deliver warehouse tables with consistent deltas, the maintenance jobs needed to be processed in order in each operator. With continuous delta batches in the input delta stream, the operator execution is deployed in the following three types, depending on how the internal/external state is accessed.

- For operators that only write or install updates to external state, the operator execution on each delta batch can be wrapped into a

transaction. Multiple transactions could be instantiated for continuous incoming delta batches and executed simultaneously while these transactions have to commit in order, which is the same as the sequence in which the maintenance jobs are created. Transactions protect from system failures, e.g. the external state would not be inconsistent in case a system crash occurs in the middle of one operator execution with partial updates. In Figure 3.1, such operators can be op_3 or op_4 which continuously update the target warehouse tables. Having multiple concurrent transaction executions on incoming delta batches with a strict commit order is useful to increase the throughput.

- For operators or more complex user-defined procedures which could both read and write the same states, transactions run serially for incoming delta batches. For example, op_4 calculates average stock price, which needs to read the stock prices installed by the transaction executions on the preceding delta batches.

- For operators that do not access any external state or probably read a private state which is rarely mutated by other applications, no transaction is needed for the operator execution. The drawback of running a transformation operator in one transaction is that the output deltas will only be visible to downstream operator when the transaction commits. To execute operators like filter or surrogate-key-lookup (op_2), no transactions are issued. The output delta batches of these operators are generated in a tuple-by-tuple fashion and can be immediately processed by subsequent operators, thus increasing the throughput of flow execution.

- A more complicated case is that multiple separate operators could access the same shared (external) state. Thus, additional scheduling and coordination of operator executions are needed, which is detailed in Section 3.5.

3.2 Incremental ETL Pipeline

As introduced before, a sequence of query arrivals force our ETL maintenance flow to work on a list of running, chained maintenance jobs (called *maintenance job chain*), each of which brings relevant warehouse tables to a consistent state demanded by a specific query. We address the efficiency

challenge of ETL maintenance flow execution in this section. We exploit pipeline parallelism and proposed an idea of *incremental ETL pipeline*.

In more detail, we define three status of a maintenance job: *pending*, *in-progress* and *finished*. When the system initially starts, a pending maintenance job is constructed and put in an empty maintenance job chain. Before any query arrives, all captured source delta tuples are tagged with the id of this job. With the event of a query arrival, the status of this pending job is changed to in-progress and all delta tuples with this job id are grouped to a delta batch as input. A new pending maintenance job is immediately constructed and appended to the end of the job chain, which is used to mark subsequent incoming source deltas with this new job id. The job ids contained in the tuples from delta batches are used to distinguish different maintenance jobs executed in the incremental ETL pipeline.

The ETL pipeline is a runtime implementation of the dataflow graph where each node runs in a single, non-terminating thread (*operator thread*[1]) and each edge $e \in E$ is an in-memory pipe used to transfer data from its provider operator thread to the consumer operator thread. Each transformation operator contains a pointer which iterates through those job ids in the maintenance job chain.

An operator thread continuously processes tuples from incoming delta batches and only blocks if its input pipe is empty or when it points at a job id with pending as its job status. When the job status changes to in-progress (e.g. when a query occurs), the blocked operator thread wakes up and uses the current job id to fetch delta tuples with matching job id from its input pipe. When an operator thread finishes the current maintenance job, it re-initializes its local state (e.g. cache, local variables) and tries to fetch the next (in-progress) maintenance jobs by moving its pointer along the job chain. In this way, we construct a maintenance job pipeline where every operator thread works on its own job (even for blocking operators, e.g. sort, as well). The notion of pipelining in our case is defined at job level instead of row level. However, row-level pipelining still occurs when threads of multiple adjacent operators work on the same maintenance job.

Figure 3.2 illustrates a state where the ETL pipeline is flushed by four maintenance jobs ($m_1 \sim m_4$). These jobs are triggered by either queries or update overload[2]. At the end of this maintenance job chain exists a

[1]As defined in Section 3.1, there are three deployment types of operator which decide whether the operator execution in this thread is wrapped in a transaction or not.

[2]We also introduce system-level maintenance jobs which are generated when the size of an input delta stream exceeds a certain threshold without any necessary query arrival. This partially hides maintenance overhead from query response time, thus

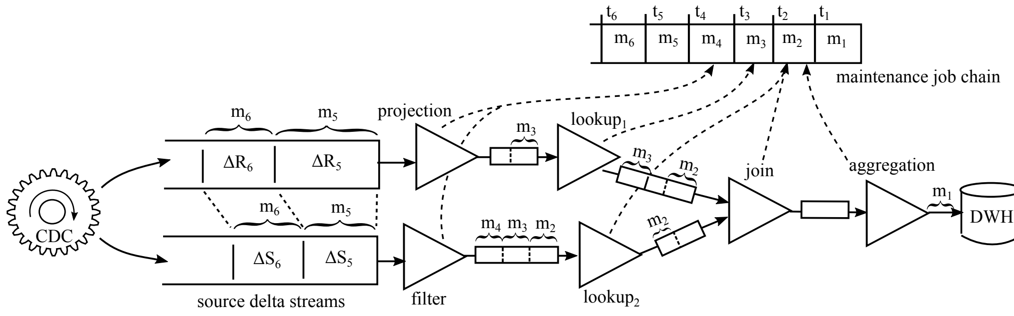


Figure 3.2: Incremental ETL Pipeline

pending m_6 job used to assign the id of m_6 to later captured deltas. In this example, the downstream aggregation thread has delivered target deltas of m_1 to the warehouse and blocks when it tries to work on m_2 since there is still no output from its preceding (blocking) join thread. The lookup$_2$ in the bottom join branch is still working on m_2 due to slow speed or large input size while the lookup$_1$ in the upper join branch is generating output deltas of m_3. However, the deltas with the id of m_3 in the input pipe are invisible to the join thread until it finishes m_2. Besides, a large pile-up exists in the input pipe of lookup$_2$ and more CPU cycles could be needed for it to solve transient overload. From this example, we see that our incremental ETL pipeline is architected and designed to handle continuously incoming maintenance jobs simultaneously and efficiently.

3.3 The Consistency Model

In this section, we introduce the notion of consistency which our work is building on. For simplicity, let us assume that an ETL flow f is given with one source table I and one target warehouse table S as sink. With an arrival of a query Q_i at point in time t_i, the maintenance job is denoted as m_i and the delta batch in the source delta stream for source table I is defined as $\Delta_{m_i} I$. After one run of maintenance flow execution on $\Delta_{m_i} I$, the final delta batch for updating the target table S is defined as follows:

$$\Delta_{m_i} S = f(\Delta_{m_i} I)$$

Given an initial state S_{old} for table S, the correct state that is demanded by the first incoming query Q_1 is derived by updating (denoted as $\uplus$) the initial state S_{old} with the final delta batch $\Delta_{m_1} S$. As defined above, $\Delta_{m_1} S$ is

shrinking the synchronization delay for answering a late query.

calculated from the source deltas $\Delta_{m_1} I$ which is captured from the source-local transactions committed before the arriving time of Q_1, i.e. t_1.

$$
\begin{aligned}
S_{m_1} &\equiv S_{old} \uplus \Delta_{m_1} S \equiv S_{old} \uplus f(\Delta_{m_1} I) \\
S_{m_2} &\equiv S_{m_1} \uplus \Delta_{m_2} S \equiv S_{old} \uplus \Delta_{m_1} S \uplus \Delta_{m_2} S \equiv S_{old} \uplus f(\Delta_{m_1} I) \uplus f(\Delta_{m_2} I) \\
&\ldots. \\
S_{m_i} &\equiv S_{m_{i-1}} \uplus \Delta_{m_1} S \\
&\equiv S_{m_{i-2}} \uplus \Delta_{m_{i-1}} S \uplus \Delta_{m_i} S \\
&\ldots \\
&\equiv S_{old} \uplus \Delta_{m_1} S \uplus \Delta_{m_2} S \ldots \uplus \Delta_{m_{i-1}} S \uplus \Delta_{m_i} S \\
&\equiv S_{old} \uplus f(\Delta_{m_1} I) \uplus f(\Delta_{m_2} I) \ldots \uplus f(\Delta_{m_{i-1}} I) \uplus f(\Delta_{m_i} I)
\end{aligned}
$$

Therefore, we define that a snapshot of table S_{m_i} is *consistent* for the query Q_i if S_{m_i} is contiguously updated by final delta batches from preceding maintenance jobs ($m_1 \sim m_i$) before the submission time of Q_i and does not include any update from fast-finished succeeding jobs (e.g. m_{i+1}), which excludes non-repeatable read/phantom read anomalies).

An example is depicted in Figure 3.3. The CDC process is continuously running and sending captured deltas from OLTP sources (e.g. transaction log) to the ETL maintenance flow which propagates updates to warehouse tables on which OLAP queries are executed. In our example, the CDC process has successfully extracted delta tuples of three committed transactions T_1, T_2 and T_3 from the transaction log files and buffered them in the DPA of the ETL maintenance flows. The first query Q_1 occurs at

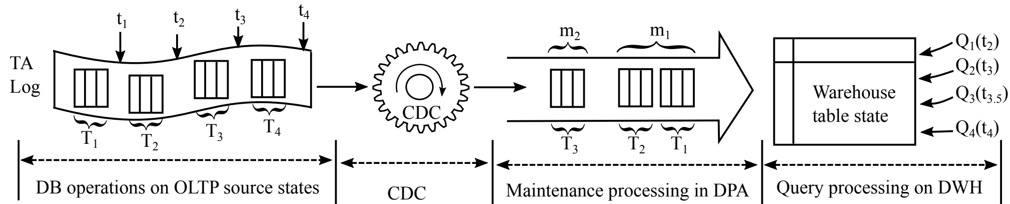


Figure 3.3: Consistency Model Example

the warehouse side at time t_2. The execution of Q_1 is first suspended until its relevant warehouse tables are updated by maintenance flows using available captured deltas of T_1 and T_2 which are committed before t_2. The delta tuples of T_1 and T_2 are grouped together as an input delta batch with the id of the maintenance job m_1. Once m_1 is finished, Q_1 is resumed

and sees an up-to-date snapshot. The execution of the second query Q_2 (at t_3) forces the warehouse table state to be upgraded with another maintenance job m_2 with only source deltas derived from T_3. Note that, due to serializable snapshot isolation mechanisms, the execution of Q_1 always uses the same snapshot that is taken from the warehouse tables refreshed with the final delta batch of m_1, and will not be affected by the new state that is demanded by Q_2. The third query Q_3 occurs at $t_{3,5}$ preceding the commit time of T_4. Therefore, no additional delta needs to be propagated for answering Q_3 and it uses the same snapshot as Q_2.

We assume that the CDC is always capable of delivering up-to-date changes to the DPA for real-time analytics. However, this assumption normally does not hold in reality and maintenance anomalies might occur in this situation as addressed by Zhuge et al. [ZGMHW95]. In Figure 3.3, there is a CDC delay between the recording time of T_4's delta tuples in the transaction log and their occurrence time in the DPA of the ETL flow. The occurrence of the fourth query Q_4 arriving at t_4 requires a new warehouse state updated by the deltas of T_4 which are still not available in the DPA. We provide two realistic options here to compensate for such potential CDC implementation. The first option is to relax the query consistency of Q_4 and let it use the same snapshot as Q_2 and Q_3. OLAP queries can usually tolerate small delays in updates and a "tolerance window"can be set (e.g., 30 seconds or 2 minutes) to allow scheduling the query without having to wait for all updates to arrive. This tolerance window could be set arbitrarily. Another option is to force maintenance processing to hang on until the CDC has successfully delivered all required changes to the DPA with known scope of input deltas for answering Q_4. With these two options, we continue with introducing our workload scheduler and incremental ETL pipeline.

3.4 Workload Scheduler

As we defined the consistency notion in the previous section, the suspended execution of any incoming query resumes only if relevant tables are refreshed by a corresponding final delta batch. Updating warehouse tables is normally done by the last (sink) operator in our incremental ETL pipeline and transactions are run to permanently install updates from multiple delta batches into warehouse tables. We denote the transactions running in the last sink operator thread as *sink transactions* (ST). In this section, we focus on our workload scheduler, which is used to orchestrate the execution of sink transactions and OLAP queries. Scheduling

constraints are introduced which deliver an execution order of begin and commit actions among sink transactions and OLAP queries.

Recall that an event of a query arrival Q_i immediately triggers the creation of a new maintenance job m_i, which updates the warehouse state for Q_i. The execution of Q_i is suspended until m_i is completed in the ST_i (i.e. the *i*-th transaction execution of ST commits successfully with its commit action $c(ST_i)$). Query Q_i is later executed in a transaction as well in which the begin action (denoted as $b(Q_i)$) takes a snapshot of the new warehouse state changed by ST_i. Therefore, the first integrity constraint enforced by our workload scheduler is $t(c(ST_i)) < t(b(Q_i))$ which means that ST_i should be committed before Q_i starts.

With arrivals of a sequence of queries {Q_i, Q_{i+1}, Q_{i+2}, ...}, a sequence of corresponding sink transactions {ST_i, ST_{i+1}, ST_{i+2}, ...} are run for corresponding final delta batches. Note that, once the $b(Q_i)$ successfully happens, the query Q_i does not block its successive sink transaction ST_{i+1} for consistency control since the snapshot taken for Q_i is not interfered by ST_{i+1}. Hence, {ST_i, ST_{i+1}, ST_{i+2}, ...} can run concurrently and commit in order while each $b(Q_i)$ is aligned with the end of its corresponding $c(ST_i)$ into {$c(ST_i)$, $b(Q_i)$, $c(ST_{i+1})$, ...}. However, only with the first constraint, the serializability property is still not guaranteed since the commit action $c(ST_{i+1})$ of a simultaneous sink transaction execution might precede the begin action $b(Q_i)$ of its preceding query. For example, after ST_i is committed, the following ST_{i+1} might be executed and committed so fast that Q_i has not yet issued the begin action. The snapshot now taken for Q_i includes rows updated by deltas occurring later than Q_i's submission time, which incurs non-repeatable/phantom read anomalies. In order to avoid these issues, the second integrity constraint is $t(b(Q_i)) < t(c(ST_{i+1}))$. This means that each sink transaction is not allowed to commit until its preceding query has successfully begun. Therefore, a serializable schedule can be achieved if the integrity constraint $t(c(ST_i)) < t(b(Q_i)) < t(c(ST_{i+1}))$ is not violated. The warehouse state is incrementally maintained by a sequence of consecutive sink transactions in response to the consistent snapshots required by incoming queries.

Figure 3.4 illustrates the implementation of the workload scheduler. An internal queue called *ssq* is introduced for a serializable schedule of sink and query transactions. Each element *e* in *ssq* represents the status of a corresponding transaction and serves as a waiting point to suspend the execution of its transaction. We also introduced the three levels of query consistency (i.e. *open*, *closed* and *complete*) defined in [GJ11] in our work to identify the status of the sink transaction (see Subsection 2.4.3). At any time there is always one and only one open element stored at the end of

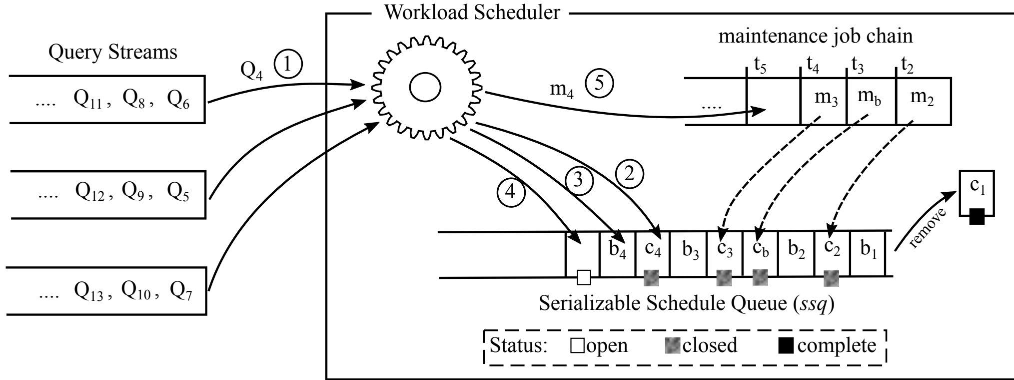


Figure 3.4: Scheduling Sink Transactions and OLAP Queries

ssq to indicate an open sink transaction (which is ST_4 in this example). Once a query (e.g. Q_4) arrives at the workload scheduler①, the workload scheduler first changes the status of the last element in *ssq* from open to closed. This indicates that the maintenance job for a pending ST_4 has been created and the commit c_4 of ST_4 should wait for the completion of this *ssq* element②. Furthermore, a new element b_4 is pushed into *ssq* which suspends the execution of Q_4 before its begin action③. Importantly, another new open element is created and put at the end of *ssq* to indicate the status of a subsequent sink transaction triggered by the following incoming query (e.g. Q_5)④. ST_4 is triggered to be started afterwards⑤. When ST_4 is done and all the deltas have arrived at the warehouse site, it marks its *ssq* element c_4 as complete and keeps waiting until c_4 is removed from *ssq*. Our workload scheduler always checks the status of the head element of *ssq*. Once its status is changed from closed to complete, it removes the head element and notifies the corresponding suspended transaction to continue with subsequent actions. In this way, the commit of ST_4 would never precede the beginning of Q_3 which takes a consistent snapshot maintained by its preceding maintenance transactions {ST_2, ST_b[3], ST_3}. Besides, Q_4 begins only after ST_4 has been committed. Therefore, the constraints are satisfied and a serializable schedule is thereby achieved.

[3]A system-level maintenance job is constructed and executed by the ST_b transaction, as certain source delta stream exceeds a pre-defined threshold.

3.5 Operator Thread Synchronization and Coordination

In the Section 3.2, we see that the incremental ETL pipeline is capable of handling multiple maintenance jobs simultaneously. However, for those operator threads which read and write the same intermediate staging tables or warehouse dimension tables in the same pipeline, inconsistencies can still arise in the final delta batch. In this section, we first address inconsistency anomalies in two cases: incremental join and slowly changing dimensions. After that, we introduce a new concept of *consistency zones* which is used to synchronize/coordinate operator threads for consistent target deltas. In the end, we discuss the options to improve the efficiency of an incremental ETL pipeline with consistency zones.

3.5.1 Pipelined Incremental Join

An incremental join is a logical operator which takes the deltas (insertions, deletions and updates) on two join tables as inputs and calculates target deltas for previously derived join results. In [BJ10], a delta rule[4] was defined for incremental joins (shown as follows). Insertions on table R are denoted as ΔR and deletions as ∇R. Given the old state of the two join tables (R_{old} and S_{old}) and corresponding insertions (ΔR and ΔS), new insertions affecting previous join results can be calculated by first identifying matching rows in the mutual join tables for the two insertion sets and further combining the newly incoming insertions found in ($\Delta R \bowtie \Delta S$). The same applies to detecting deletions.

$$\Delta(R \bowtie S) \equiv (\Delta R \bowtie S_{old}) \cup (R_{old} \bowtie \Delta S) \cup (\Delta R \bowtie \Delta S)$$

$$\nabla(R \bowtie S) \equiv (\nabla R \bowtie S_{old}) \cup (R_{old} \bowtie \nabla S) \cup (\nabla R \bowtie \nabla S)$$

For simplicity, we use the symbol Δ to denote all insertions, deletions and updates in this chapter. Hence, the first rule is enough to represent incremental join with an additional join predicate (R.action = S.action) added to $(\Delta R \bowtie \Delta S)$ where action can be insertion I, deletion D or update U.

We see that a logical incremental join operator is mapped to multiple physical operators, i.e. three join operators plus two union operators. To implement this delta rule in our incremental ETL pipeline, two

[4]Updates are treated as deletions followed by insertions in this rule.

tables R_{old} and S_{old} are materialized in the staging area during historical load and two extra update operators (denoted as $\uplus$) are introduced. One $\uplus$ is used to gradually maintain the staging table S_{old} using the deltas ($\Delta_{m_1}S, \Delta_{m_2}S, ...\Delta_{m_{i-1}}S$) from the executions of preceding maintenance jobs ($m_1, m_2, ..., m_{i-1}$) to bring the join table S_{old} to *consistent* state $S_{m_{i-1}}$ for $\Delta_{m_i}R$:

$$S_{m_{i-1}} = S_{old} \uplus \Delta_{m_1}S ... \uplus \Delta_{m_{i-1}}S$$

Another update operator $\uplus$ performs the same on the staging table R_{old} for $\Delta_{m_i}S$. Therefore, the original delta rule is extended in the following based on the concept of our maintenance job chain.

$$\begin{aligned} \Delta_{m_i}(R \bowtie S) &\equiv (\Delta_{m_i}R \bowtie S_{m_{i-1}}) \cup (R_{m_{i-1}} \bowtie \Delta_{m_i}S) \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S) \\ &\equiv (\Delta_{m_i}R \bowtie (S_{old} \uplus \Delta_{m_{1\sim(i-1)}}S)) \cup ((R_{old} \uplus \Delta_{m_{1\sim(i-1)}}R) \bowtie \Delta_{m_i}S) \\ &\quad \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S) \end{aligned}$$

The deltas $\Delta_{m_i}(R \bowtie S)$ of job m_i are considered as *consistent* only if the update operators have completed job $m_{(i-1)}$ on two staging tables before they are accessed by the join operators. However, our ETL pipeline only ensures that the maintenance job chain is executed in sequence in each operator thread. Inconsistency can occur when directly deploying this extended delta rule in our ETL pipeline runtime. This is due to concurrent executions of join and update operators on the same staging table for different jobs. We use a simple example (see Figure 3.5) to explain the

Customer (refreshed by m_1)

id	name	company
1	bob	IBM
2	mary	SAP

Company (refreshed by m_1)

name	nation
IBM	USA

Δ(Customer ⋈ Company)

job	action	value
m_2	—	∅
m_3	I	(3, 'jack', 'HP', 'USA') (2, 'mary', 'SAP', 'Germany')
m_4	I	(4, 'peter', 'SAP', 'Germany')

ΔCustomer

job	action	value
m_2	—	—
m_3	I	(3, 'jack', 'HP')
m_4	I	(4, 'peter', 'SAP')

ΔCompany

job	action	value
m_2	I	('HP', 'USA')
m_3	I	('SAP', 'Germany')
m_4	—	—

Incorrect Δ(Customer ⋈ Company):
$(\Delta_{m3}Customer \bowtie Company_{m1}) \cup (Customer_{m4} \bowtie \Delta_{m3}Company) \cup (\Delta_{m3}Customer \bowtie \Delta_{m3}Company)$

job	action	value
m_3	I	(2, 'mary', 'SAP', 'Germany') (4, 'peter', 'SAP', 'Germany')

Figure 3.5: Anomaly Example for Pipelined Incremental Join

potential anomaly. The two staging tables *Customer* and *Company* are depicted at the left-upper part of Figure 3.5 which both have been updated by deltas from m_1. Their input delta streams are shown at left-bottom part and each of them contains a list of tuples in the form of (job, action, value)

which is used to store insertion-/deletion-/update-delta sets (only insertions with action I are considered here) for each maintenance job. Logically, by applying our extended delta rule, consistent deltas Δ(Customer $\ltimes$ Company) would be derived which are shown at the right-upper part. For job m_3, a matching row ('HP', 'USA') can be found in the company table for a new insertion (3, 'jack', 'HP') on the customer table after the company table was updated by the preceding job m_2. With another successful row-matching between Δ_{m_3}Company and Customer$_{m_2}$, the final deltas are complete and correct.

However, since at runtime, each operator thread runs independently and has different execution latencies for inputs of different sizes, an inconsistent case can occur which is shown at the right-bottom part. Due to various processing costs, the join operator Δ_{m_3}Customer$\ltimes$Company$_{m_1}$ has already started before the update operator completes m_2 on the company table, which mistakenly missed the matching row ('HP', 'USA') from m_2. And the other join operator Customer$_{m_4}\ltimes\Delta_{m_3}$ Company accidentally reads a phantom row (4, 'peter', 'SAP') from the maintenance job m_4 that is accomplished by the fast update operator on the customer table. This anomaly is caused by a pipeline execution without synchronization of read-/write-threads on the same staging table.

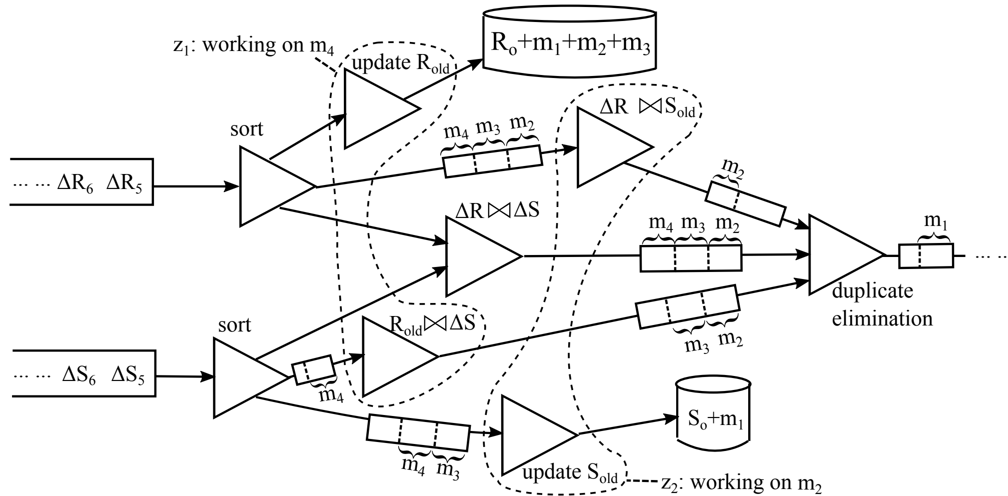


Figure 3.6: Pipelined Incremental Join with Consistency Zones

To address this problem, we propose a *pipelined incremental join* for the maintenance job chain. It is supported by newly defined *consistency zones* and an extra duplicate elimination operator. Figure 3.6 shows the imple-

mentation of our pipelined incremental join[5]. In a consistency zone, operator thread executions are synchronized on the same maintenance job and processing of a new maintenance job is not started until all involving operator threads have completed the current one. This can be implemented by embedding a *cyclic barrier*(*cb*) (Java) object in all covered threads. Each time a new job starts in a consistency zone, this *cb* object sets a local count to the number of all involved threads. When a thread completes, it decrements the local count by one and blocks until the count becomes zero. In Figure 3.6, there are two consistency zones: z_1(update-R$_{old}$, R$_{old}$ $\bowtie$ ΔS) and z_2(ΔR$\bowtie$S$_{old}$, update-S$_{old}$), which group together all the threads that read and write the same staging table. The processing speeds of both threads in z_1 are very similar and fast, so both of them are currently working on m_4 and there is no new maintenance job buffered in any of the in-memory pipes of them. However, even though the original execution latency of the join operator thread ΔR$\bowtie$S$_{old}$ is low, it has to be synchronized with the slow operator update-S$_{old}$ on m_2 and a pile-up of maintenance jobs ($m_{2\sim4}$) exists in its input pipe. It is worth to note that a strict execution sequence of two read-/write threads is not required in a consistency zone (i.e. update-R$_{old}$ does not have to start only after R$_{old}$ $\bowtie$ ΔS completes to meet the consistency requirement R$_{m_{i-1}}$ $\bowtie$ Δ_{m_i}S). In case R$_{m_{i-1}}$ $\bowtie$ Δ_{m_i}S reads a subset of deltas from m_i (in R) due to concurrent execution of update-R$_{m_{i-1}}$ on m_i, duplicates will be deleted from the results of Δ_{m_i}R$\bowtie$$\Delta_{m_i}$S by the downstream duplicate elimination operator. Without a strict execution sequence in consistency zones, involved threads can be scheduled on different CPU cores for performance improvement. Furthermore, even though two consistency zones finish maintenance jobs in different paces, this duplicate elimination operator serves as a *Merger* and only reads correct input deltas for its current maintenance job, which is m_2 in the example.

3.5.2 Pipelined Slowly Changing Dimensions

In data warehouses, slowly changing dimension (SCD) tables need to be maintained which change over time. The physical implementation depends on the type of SCD (three SCD types are defined in [KC04]). For example, SCDs of type 2 are history-keeping dimensions where rows comprising the same business key represent a history of one entity while each row has a unique surrogate key in the warehouse and was valid in a cer-

[5] The two sort operators are just required for merge join and can be omitted if other join implementations are used.

tain time period (from start date to end date and the current row version has the end date null). With a change occurring in the source table of a SCD table, the most recent row version of the corresponding entity (end date is null) is updated by replacing the null value with the current date and a new row version is inserted with a new surrogate key and a time range (current date $\sim$ null). In the fact table maintenance flow, the surrogate key of this current row version of an entity is looked up as a foreign key value in the fact table.

Assume that the source tables that are used to maintain fact tables and SCDs reside in different databases. A globally serializable schedule S of the source actions on these source tables needs to be replayed in ETL flows for strong consistency in data warehouses [ZGMW96]. Otherwise, a consistency anomaly can occur which will be explained in the following (see Figure 3.7).

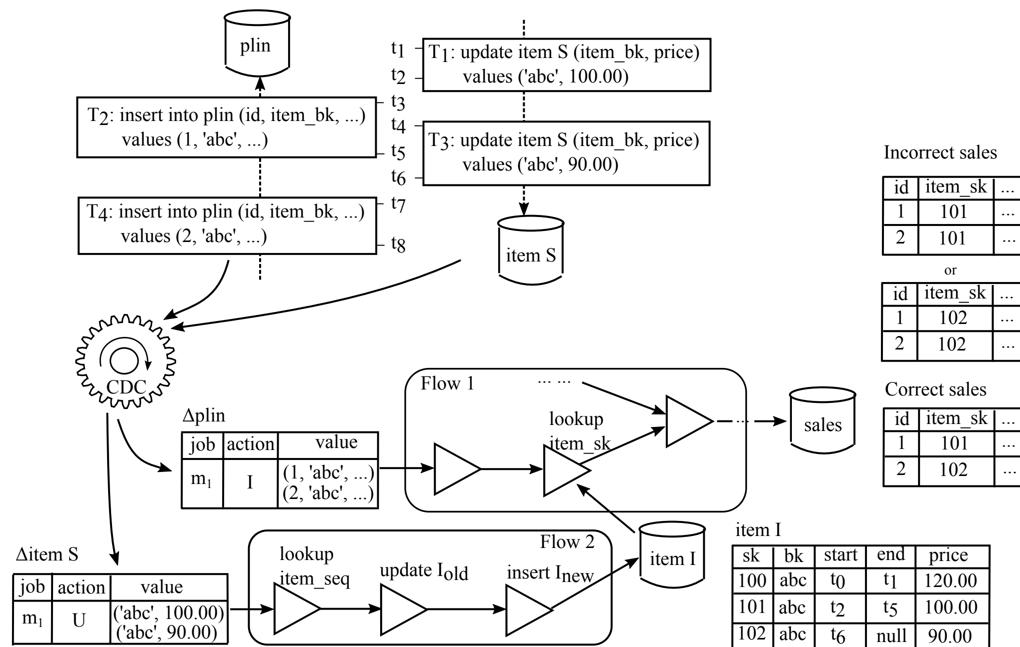


Figure 3.7: Anomaly Example for ETL Pipeline Execution without Coordination

At the upper-left part of Figure 3.7, two source tables: *plin* and *item-S* are used as inputs for a fact table maintenance flow (Flow 1) and a dimension maintenance flow (Flow 2) to refresh warehouse tables *sales* and *item-I*, respectively. Two source-local transactions T_1 (start time: t_1 $\sim$ commit time: t_2) and T_3 ($t_4$$\sim$$t_6$) have been executed on *item-S* to update the price

attribute of an item with business key ('abc') in one source database. Two additional transactions T_2 ($t_3 \sim t_5$) and T_4 ($t_7 \sim t_8$) have been also completed in a different database where a new state of source table *plin* is affected by two insertions sharing the same business key ('abc'). Strong consistency of the warehouse state can be reached if the globally serializable schedule S: $T_1 \leftarrow T_2 \leftarrow T_3 \leftarrow T_4$ is also guaranteed in ETL pipeline execution. A consistent warehouse state has been shown at the bottom-right part of Figure 3.7. The surrogate key (101) found for the insertion (1, 'abc', ...) is affected by the source-local transaction T_1 on *item-S* while the subsequent insertion (2, 'abc', ...) will see a different surrogate key (102) due to T_3. However, the input delta streams only reflect the local schedules S_1: $T_1 \leftarrow T_3$ on *item-S* and S_2: $T_2 \leftarrow T_4$ on *plin*. Therefore, there is no guarantee that the global schedule S will be correctly replayed since operator threads run independently without coordination. For example, at time t_9, a warehouse query occurs, which triggers an immediate execution of a maintenance job m_1 that brackets T_2 and T_4 together on *plin* and groups T_1 and T_3 together on *item-S*. Two incorrect states of the *sales* fact table have been depicted at the upper-right part of the figure. The case where *item_sk* has value 101 twice corresponds to an incorrect schedule: $T_1 \leftarrow T_2 \leftarrow T_4 \leftarrow T_3$ while another case where *item_sk* has value 102 twice corresponds to another incorrect schedule: $T_1 \leftarrow T_3 \leftarrow T_2 \leftarrow T_4$. This anomaly is caused by an uncontrolled execution sequence of three read-/write-operator threads: *item_sk-lookup* in Flow 1 and *update-*I_{old} and *insert-*I_{new} in Flow 2.

To achieve a correct globally serializable schedule S, the CDC component should take the responsibility of rebuilding S by first tracking start or commit timestamps of source-local transactions[6], mapping them to global timestamps and finally comparing them to find out a global order of actions. In addition, the execution of relevant operator threads needs to be coordinated in this global order in the incremental ETL pipeline. Therefore, another type of *consistency zone* is introduced here.

Before we introduce our new consistency zone for our *pipelined SCD*, it is worth to note that the physical operator that is provided by the current ETL tool to maintain SCDs does not fulfill the requirement of the SCD (type 2) in our case. To address this, we simply implement SCD (type 2) using *update-*I_{old} followed by *insert-*I_{new}. These two operator threads need to be executed in an atomic unit so that queries and surrogate key lookups will not see an inconsistent state or fail when checking a lookup condition. Another case that matters is that the execution of Flow 1 and

[6]Execution timestamps of in-transaction statements have to be considered as well, which is omitted here.

Flow 2 mentioned previously is not performed strictly in sequence in a disjoint manner. Instead of using flow coordination for strong consistency, all operators from the two flows (for fact tables and dimension tables) are merged into a new big flow where the atomic unit of *update-I_{old} insert-I_{new}* operator threads can be scheduled with the *item_sk-lookup* operator thread at a fine-grained operator level.

Our approach for pipeline coordination used in pipelined SCD is illustrated in Figure 3.8. We first explain how the CDC process can help rebuild the global schedule S. Recall that a maintenance job is constructed when a query is issued or when the size of any input delta stream exceeds a threshold (see Section 3.2). We refine the maintenance job into multiple internal, fine-grained *tasks* whose construction is triggered by a commit action of a source-local transaction affecting the source table of a SCD. As shown in Figure 3.8, ① the CDC continuously puts those cap-

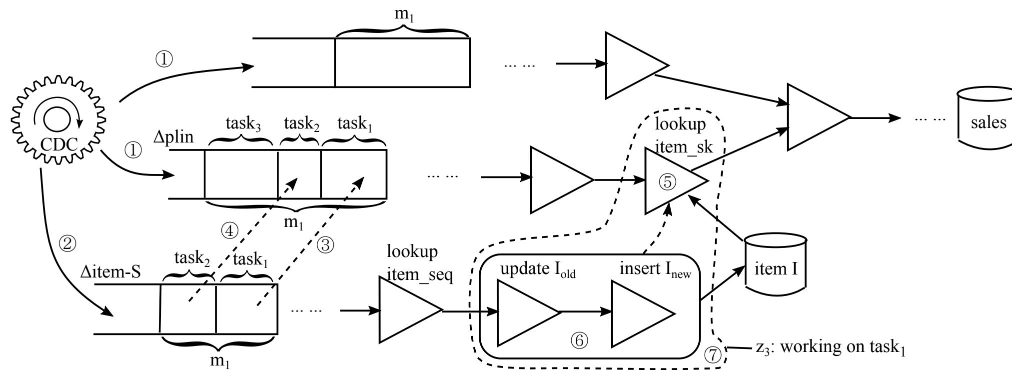


Figure 3.8: Pipelined SCD with Consistency Zone

tured source deltas into the input delta streams (one is Δ*plin*) of the fact table maintenance flow. At this time, a source-local update transaction commits on *item-S*, which creates a task_1 and comprises the delta tuples derived from this update transaction ②. This immediately creates another task_1 in the input delta stream Δ*plin* which contains all current available delta tuples ③. This means that all source-local, update transactions belonging to the task_1 in Δ*plin* have committed before the task_1 of Δ*item-S*. With a commit of the second update transaction on source table *item-S*, two new task_2 are created in both input delta streams ④. When a query is issued at a later time, a new m_1 is constructed which contains $\text{task}_{1\sim2}$ on Δ*item-S* and $\text{task}_{1\sim3}$ on Δ*plin* (delta tuples in task_3 commit after the task_2 in Δ*item-S*). During execution on m_1, a strict execution sequence between the atomic unit of *update-I_{old}* and *insert-I_{new}* and the *item_sk-lookup* is forced for each task_i in m_1. The *update-I_{old}* and *insert-I_{new}* have to wait until

the *item_sk-lookup* finishes task$_1$ ⑤ and the *item_sk-lookup* cannot start to process task$_2$ until the atomic unit completes task$_1$ ⑥. This strict execution sequence can be implemented by the (Java) *wait/notify* methods as a provider-consumer relationship. Furthermore, in order to guarantee the atomic execution of both *update-*I_{old} and *insert-*I_{new} at task level, (Java) *cyclic barrier* can be reused here to let *update-*I_{old} wait to start a new task until *insert-*I_{new} completes the current one ⑥. Both thread synchronization and coordination are covered in this consistency zone ⑦.

3.5.3 Discussion

In several research efforts on operator scheduling, efficiency improvements can be achieved by cutting a data flow into several sub-flows. In [CÇR+03], one kind of sub-flow called *superboxes* are used to group operators into batches in order to reduce the scheduling overhead. Authors of [KVS13] use similar sub-flows (*strata*) to exploit pipeline parallelism at certain level. However, in our scenario, the operators that are involved in a sub-flow are normally connected through data paths. As described in the previous sections, consistency zones can have operator threads scheduled together without any connecting data path. This would increase the complexity of general scheduling algorithms that tend to improve pipeline performance and normally address bottleneck operators with execution latency as $max(time(op_i))$. A pipeline that was previously efficient can be slowed down dramatically when one of its operator is bound with a very slow operator in a consistency zone, which further increases the $max(time(op_i))$.

The efficiency of an incremental ETL pipeline with consistency zones can be improved if the data storage supports multi-version concurrency control, where reads do not block writes and vice versa. Therefore, a fast update operator on a staging table will not be blocked by a slow join operator which reads rows using version number (possibly maintenance job id in our case). However, in another case, a fast join operator may still have to wait until the deltas with the desired version are made available by a slow update operator.

3.6 Experiments

In this section, we examine the performance of our incremental ETL pipeline with read-/update-heavy workloads running on three kinds of configuration settings. Furthermore, we evaluate our consistency-zone-aware

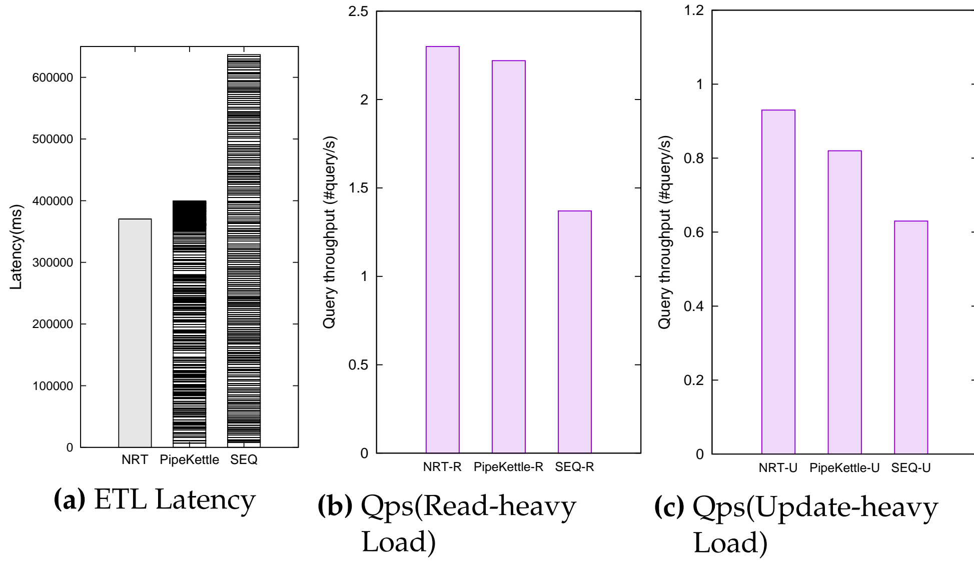
scheduling algorithm in terms of latencies of ETL jobs.

3.6.1 Incremental ETL Pipeline Performance

We used the TPC-DS benchmark [tpcb] in our experiments. Our testbed is composed of a target warehouse table *store sales* (SF 10) stored in a Postgresql [pos] (version 9.4) instance which was fine-tuned, set to serializable isolation level and ran on a remote machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM, 1TB SATA-II disk), an ETL pipeline (an integrated pipeline instance used to update item and store sales tables) running locally (Intel Core i7-4600U Processor, 2×2.10 GHz, 12GB RAM, 500GB SATA-II disk) on an extended version of Kettle (version 4.4.3) together with our workload scheduler and a set of query streams, each of which issues queries towards the remote store sales table once at a time. The maintenance flow is continuously fed by delta streams from a CDC thread running on the same node. The impact of options for handling CDC delay (see Section 3.3) was out of scope and not examined.

We first defined three configuration settings as follows. **Near Real-time (NRT)**: simulates a general near real-time ETL scenario where only one maintenance job was performed concurrently with query streams in a small time window. In this case, there is no synchronization of maintenance flow and queries. Any query can be immediately executed once it arrives and the consistency is not guaranteed. **PipeKettle**: uses our workload scheduler to schedule the execution sequence of a set of maintenance transactions and their corresponding queries. The consistency is thereby ensured for each query. Furthermore, maintenance transactions are executed using our incremental ETL pipeline. **Sequential execution (SEQ)**: is similar to **PipeKettle** while the maintenance transactions are executed sequentially using a flow instance once at a time.

Orthogonal to these three settings, we simulated two kinds of read-/update-heavy workloads as follows. **Read-heavy workload**: uses one update stream (SF 10) consisting of *purchases* (♯: 10K) and *lineitems* (♯: 120K) to refresh the target warehouse table using the maintenance flow and meanwhile issues a total of 210 queries from 21 streams, each of which has different permutations of 10 distinct queries (generated from 10 TPC-DS ad-hoc query templates, e.g. q[88]). For PipeKettle and SEQ, each maintenance job consists of 48 new purchases and 570 new lineitems in average. **Update-heavy workloads**: uses two update streams (♯: 20K & 240K) while the number of query streams is reduced to seven (totally 70 queries). Before executing a query in PipeKettle and SEQ, the number of



(a) ETL Latency

(b) Qps(Read-heavy Load)

(c) Qps(Update-heavy Load)

Figure 3.9: Throughput and Latency Comparison

deltas to be processed is 6-times larger than that in read-heavy workloads.

Figure 3.9a illustrates a primary comparison among NRT, PipeKettle and SEQ in terms of flow execution latency without query interventions. As the baseline, it took 370s for NRT to processing one update stream. The update stream was later split into 210 parts as deltas batches for PipeKettle and SEQ. It can be seen that the overall execution latency of processing 210 maintenance jobs in PipeKettle is 399s which is nearly close to the baseline due to pipelining parallelism. However, the same number of maintenance jobs is processed longer in SEQ (~650s, which is significantly higher than the others).

Figure 3.9b and 3.9c show the query throughputs measured in three settings using both read-/update-heavy workloads. Since the maintenance job size is small in read-heavy workload, the synchronization delay for answer each query is also small. Therefore, the query throughput achieved by PipeKettle (2.22 queries/s) is very close to the one in baseline NRT (2.30) and much higher than the sequential execution mode (1.37). We prove that our incremental pipeline is able to achieve high query throughput at a very high query rate. However, in update-heavy workload, the delta input size becomes larger and the synchronization delay grows increasingly, thus decreasing the query throughput in PipeKettle. Since our PipeKettle automatically triggered maintenance transactions to reduce the number of deltas buffered in the delta streams, the throughput (0.82) is still acceptable as compared to NRT(0.93) and SEQ (0.63).

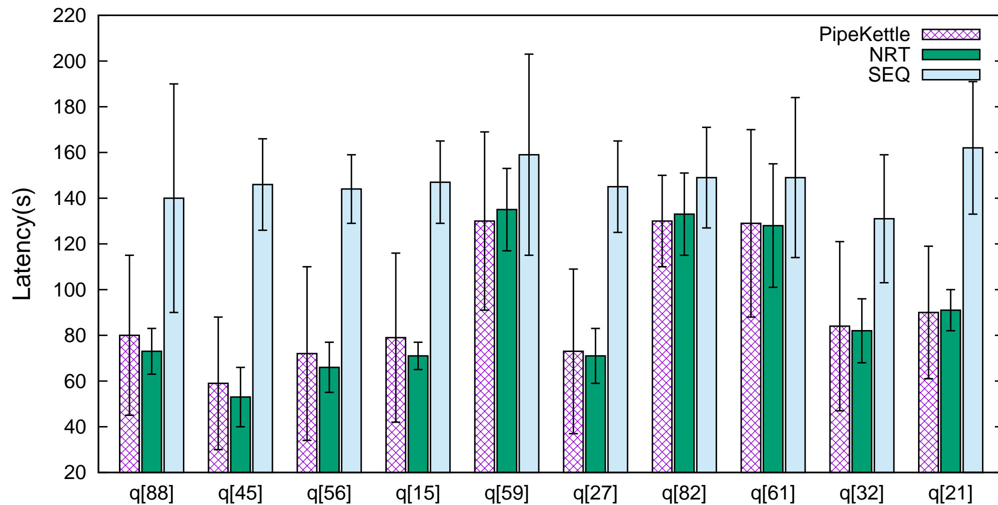


Figure 3.10: Average Latencies of 10 Ad-hoc Query Types in Read-heavy Workload

The execution latencies of 10 distinct queries recorded in read-heavy workload is depicted in Figure 3.10. Even with synchronization delay incurred by snapshot maintenance in PipeKettle, the average query latency over 10 distinct queries is approaching the baseline NRT whereas NRT does not ensure the serializability property. SEQ is still not able to cope with read-heavy workload in terms of query latency, since a query execution might be delayed by sequential execution of multiple flows. Figure 3.11 shows query latencies in update-heavy workload. With a larger number of deltas to process, each query has higher synchronization overhead in both PipeKettle and SEQ than that in read-heavy workload. However, the average query latency in PipeKettle still did not grow drastically as in SEQ since the workload scheduler triggered automatic maintenance transactions to reduce the size of deltas stored in input streams periodically. Therefore, for each single query, the size of deltas is always lower than our pre-defined batch threshold, thus reducing the synchronization delay.

3.6.2 Consistency-Zone-Aware Pipeline Scheduling

According to the scheduling algorithm *MINIMUM COST* (MC) [KVS13], an ETL workflow is divided into subflows (each of which allows pipelining operators) and the operator having the largest volume of input data is selected to execute in each subflow. In Figure 3.6, a possible fragmen-

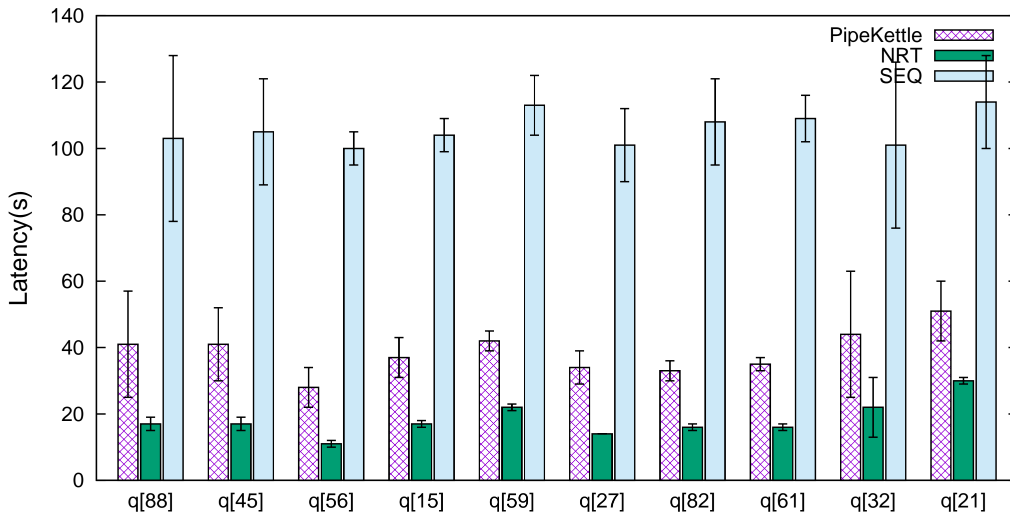
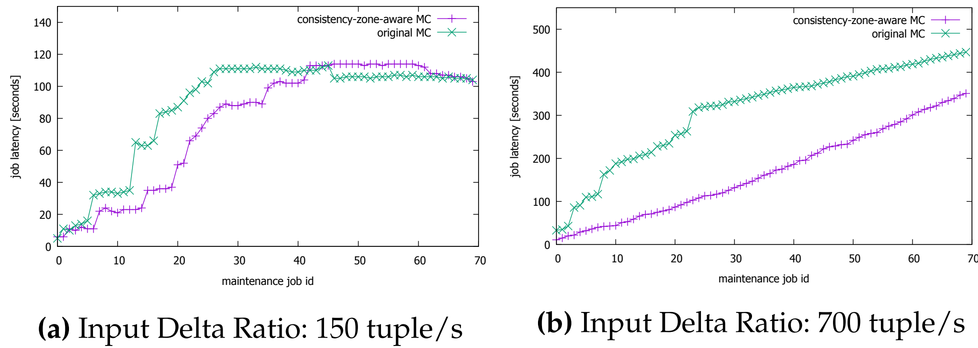


Figure 3.11: Average Latencies of 10 Ad-hoc Query Types in Update-heavy Workload

tation results in following operator groups[7]: (update-R_{old}, $\Delta R \bowtie S_{old}$), ($R_{old} \bowtie \Delta S$, update-S_{old}), ($\Delta R \bowtie S_{old}$) and (duplicate elimination). However, in incremental ETL pipeline, efficiency can degrade due to synchronized execution of threads in consistency zones. The performance of a very fast pipelined subflow can drop significantly if one of its operators hooks a separate slow operator in a consistency zone outside this subflow. The side effect of consistency zones determines that they perform like blocking operations. Hence, all operators in consistency zones should be grouped together to new subflows as z_1: (update-R_{old}, $R_{old} \bowtie \Delta S$), z_2: ($\Delta R \bowtie S_{old}$, update-S_{old}), etc., which is called consistency-zone-aware MC.

We compared the original MC and consistency-zone-aware MC and examine the latencies of maintenance jobs in two system settings where input delta streams have a low and a high input ratio, respectively (system load reaches its limit with a high input ratio). The testbed comprised a store-sales fact table (of scale factor 1) surrounding dimension tables and two staging tables materialized during historical load for pipelined incremental join. The data set is stored in a Postgresql (version 9.5) on a remote machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM). Two maintenance flows (used to maintain the store-sales fact table and the item dimension table) were merged into an incremental ETL (job) pipeline (see Figure 3.6) that ran locally (Intel Core i7-4600U Proces-

[7]blocking operations are subflows of their own.



(a) Input Delta Ratio: 150 tuple/s

(b) Input Delta Ratio: 700 tuple/s

Figure 3.12: Job Latency Comparison for Consistency-Zone-aware MC

sor, 2×2.10 GHz, 12GB RAM) in our pipeline engine which is extended from the original Pentaho Kettle engine [CBVD10]. A local CDC thread[8] simulated a low input ratio (150 tuples/s) and a high input ratio (700 tuples/s), respectively. Besides, another thread continuously issued queries to the warehouse, which triggered the constructions of maintenance jobs in random time intervals. In each setting with different scheduling policies, we collected the execution time as job latency (in seconds) for 70 maintenance jobs.

Both Figure 3.12a and Figure 3.12b show that original MC performs worse than consistency-zone-aware MC, especially under high input ratio. The reason why original MC performs slow is due to the fact that the processing cost of the lookup operator was much slower, which causes starvations of downstream $R_{old} \bowtie \Delta S$ and update-S_{old}, as they are grouped in the same subflow as defined in original MC. However, the input pipes of update-R_{old} and $\Delta R \bowtie S_{old}$ grow drastically since they block due to our consistency zone features. More time quanta were assigned to them, which is not necessary and reduces the processing quantum of the slow lookup operator. Hence, our consistency-zone-aware MC addresses this problem and groups the threads in consistency zones together to execute.

3.7 Summary

We first addressed the on-demand snapshot maintenance policy in MVCC-supported data warehouse systems using our incremental ETL

[8] ran continuously to feed the input delta streams with source deltas to update the store sales and item table.

pipeline. Warehouse tables are refreshed by continuous delta batches in a query-driven manner. A consistency model is introduced which formally defines the consistency for incoming queries. Therefore, we implemented a workload scheduler which is able to achieve a serializable schedule of concurrent maintenance flows and OLAP queries.

To accelerate both ETL jobs and query executions, we proposed our incremental ETL pipeline based on a logical computation model with a prototype implementation using an open-source ETL tool - Pentaho Kettle. Potential inconsistency anomalies for incremental join and slowly changing dimension tables were also addressed with consistency zones introduced as solutions.

The experimental results show that our approach achieves average performance close to that in traditional near real-time ETL while still guaranteeing query consistency. Besides, the job latencies with consistency-zone-aware scheduling algorithm outperform those with a traditional setting.

4 Distributed Snapshot Maintenance in Wide-Column NoSQL Databases

In this chapter, we extend the consistency model introduced in Chapter 3 to adapt to large-scale, distributed analytics. A distributed variant of our incremental ETL pipeline (called HBelt) was introduced to achieve the same "On-Demand ETL" objective in distributed warehouse systems which are built over large-scale databases (e.g., HBase). HBelt aims at providing consistent, distributed snapshot maintenance for concurrent table scans across different analytics jobs.

The remainder of this chapter is as follows. We first position our discussion background and context in Section 4.1. In Section 4.2, we introduce our HBelt system which tightly integrates a distributed ETL processing engine - Pentaho Kettle, with a large-scale data analytics platform - HBase, for data freshness, performance and consistency. Finally, experiments are conducted and discussed in Section 4.3.

4.1 Motivation

Wide-column NoSQL databases are an important class of NoSQL (Not only SQL) databases [Cat11] which scale horizontally and feature high access performance on sparse tables. With current trends towards big Data Warehouses (DWs), it is attractive to run existing business intelligence/data warehousing applications on higher volumes of data in wide-column NoSQL databases for low latency by mapping multidimensional models to wide-column NoSQL models or using additional SQL add-ons [Deh16, DBBK15, CEMK+15]. For examples, applications like retail management can run over integrated data sets stored in big DWs or in the cloud to capture current item-selling trends.

Figure 4.1 depicts a scenario where data warehousing applications are deployed on a distributed database. Online incremental ETL processes [VS09] are continuously running to propagate source data changes provi-

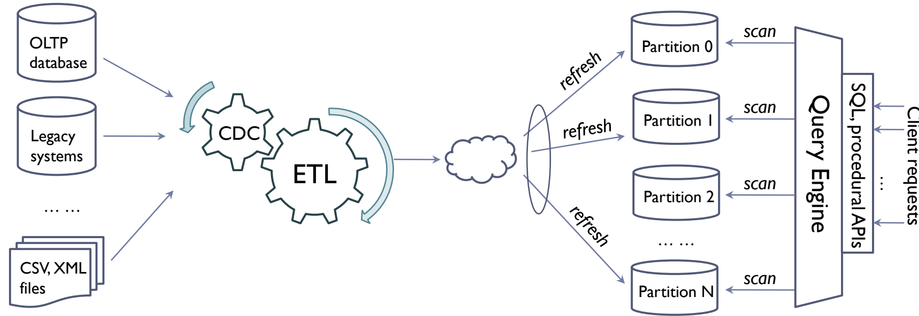


Figure 4.1: ETL Maintenance Flows for Distributed Databases

ded by a Change Data Capture (CDC) process from remote sources to the data warehouse. Distributed data partitions are refreshed using the output deltas that are calculated by ETL maintenance jobs. For warehouse systems that support Snapshot Isolation (SI) or Multi-Version Concurrency Control (MVCC) (readers do not block writers and vice versa), high throughput can be achieved in data warehousing workloads where each local scan request works on its own local snapshot and is therefore not affected by updates from concurrent maintenance jobs. We define a *distributed snapshot* as a set of local snapshots taken by the distributed scan request from an analytical query.

However, the snapshot made available in the DW is often stale, since at the moment when an analytical query is issued, the source updates (e.g. in a remote retail store) may not have been extracted and processed by the ETL process in time due to high input data volume or slow processing speed. This staleness may cause incorrect results for time-critical decision support queries. To address this problem, snapshots which are supposed to be accessed by analytical queries need to be first maintained by corresponding ETL flows to reflect source updates based on given freshness needs. Snapshot maintenance in this context means refreshing those distributed data partitions that are required by a query. Since most NoSQL databases are not ACID compliant and do not provide full-fledged distributed transaction support, snapshot may be inconsistently derived when its data partitions are updated by different, concurrent ETL maintenance jobs.

The work described in this chapter extends the work introduced in Chapter 3 by tightly integrating a wide-column NoSQL database - HBase [Geo11] with a clustered & pipelined ETL engine - Kettle [CBVD10]. The objective is to efficiently refresh HBase tables with remote source updates while a consistent snapshot is guaranteed across distributed partitions

for each scan request in analytical queries. A consistency model is defined and implemented to address so-called *distributed snapshot maintenance*. To achieve this, ETL jobs and analytical queries are scheduled in a distributed processing environment. In addition, a partitioned, incremental ETL pipeline is introduced to increase the performance of ETL (update) jobs. We validate the efficiency gain in terms of data pipelining and data partitioning using the TPC-DS benchmark [tpcb], which simulates a modern decision support system for a retail product supplier. Experimental results show that high query throughput can be achieved in HBelt when distributed, refreshed snapshots are demanded.

4.2 The HBelt System for Distributed Snapshot Maintenance

In this section, we introduce our HBelt system [QSGD15], which integrates a distributed big data store (HBase) with a distributed, pipelined data integration engine based on Kettle for real-time analytics.

Analytical queries are issued to a SQL engine running on top of an HBase store which provides only primitive operations (e.g. put, scan) to distributed table regions. In order to keep track of concurrent data changes at the source side, the state of the HBase tables that are accessed by incoming queries is maintained by multiple ETL pipeline instances. Each analytical, read-only query (comprising a set of distributed sub-requests) accesses a consistent view of data partitions that are refreshed using the latest source updates (i.e. deltas) preceding the submission time of this query. We try to reduce the maintenance delay by employing two kinds of parallel computing techniques: data partitioning and data pipelining. Therefore, the objective of HBelt is to ensure freshness, consistency and performance. The architecture is illustrated in Figure 4.2.

4.2.1 Architecture Overview

As introduced in Section 2.7, a table stored in HBase is horizontally partitioned into a set of regions with non-overlapping key ranges and distributed across multiple region servers. The current Kettle implementation (since Version 5.1) has provided a so-called HBase Output step to maintain a HBase table in a single flow instance. All calculated deltas have to go through this step to arrive in target region servers. However, since both HBase and Kettle follow a master/slave architecture, it is desirable

to utilize the essence of distributed processing from both systems for performance optimization. In HBelt, given a number of HBase region servers, the same number of redundant copies of ETL flow instances are constructed, each of which is directly running on the same node shared by a local region server. While a local region server can maintain multiple regions, only a single flow instance is dedicated to refreshing all local regions for arbitrary scan requests (e.g. one flow instance for two regions in the first region server in Figure 4.2). Creating more flow instances for regions in one region server would result in high resource utilization and extra flow synchronization overhead.

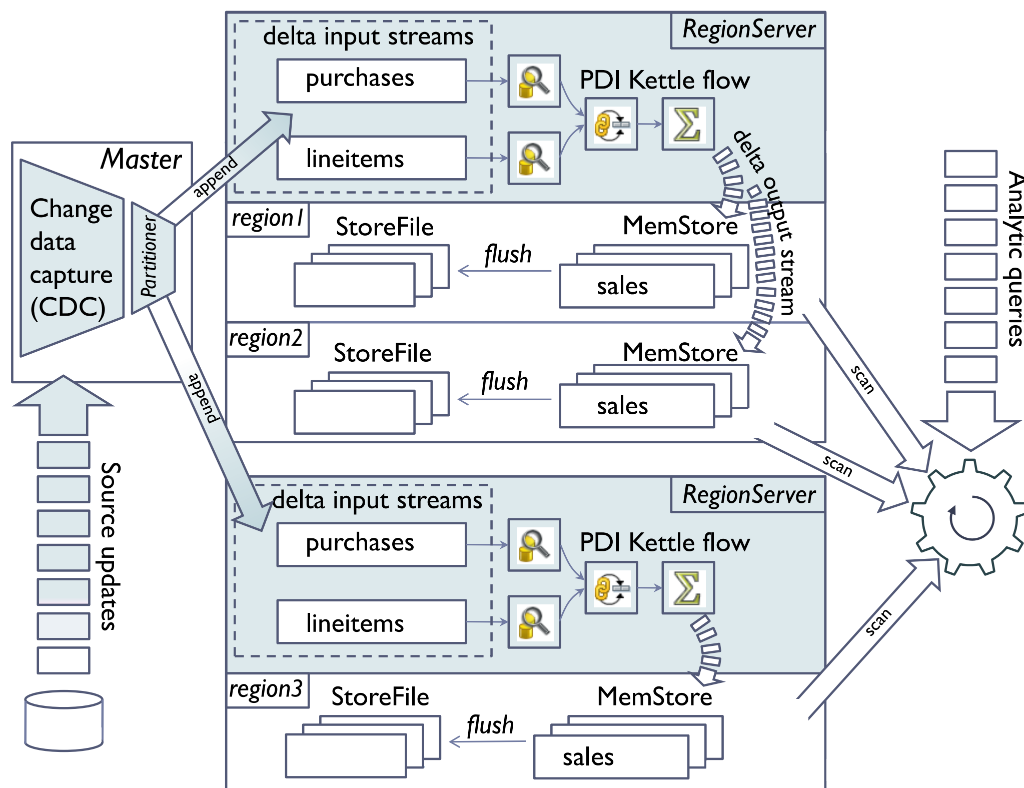


Figure 4.2: HBelt Architecture

In Figure 4.2, we use a logical ETL flow example that consists of two dimension table lookup operations, a sort-merge join and an aggregation operation. It incrementally propagates change data captured from external *purchases* and *lineitems* sources to the target *sales* table stored in HBase. In the master node (at the left side of Figure 4.2), a change data capture (CDC) process runs and uses methods like log-sniffing [KC04] or timestamps to capture the source deltas. These source deltas need to be split

in the master and affect the freshness of all the regions that are distributed in the HBase cluster.

In order to forward source delta chunks to the correct region server and assure that the target deltas calculated by subsequent ETL flow execution have to reside in the same region server, both the keys in the deltas and the key ranges of regions stored in HBase tables have to be considered. This is done by a component called *Partitioner* (a customized partitioner in Kettle). The partitioner resides in the same master node and maintains an internal meta-data structure which caches the key range and location information of all regions from the target HBase table. Given a pre-defined source-to-target key mapping, the partitioner performs range-based partitioning and returns an index indicating the correct region server number.

In this example, *purchase* rows have *purc_id* as key and both *lineitems* rows and the *sales* table share the same compound keys (*purc_id*, *item_id*). This key mapping information is used by the partitioner to distribute the source deltas of *lineitem* to the correct region servers. For a *purchases* row whose *purc_id* might span across regions in multiple region servers, copies of this *purchases* row are sent to all region servers along with *lineitems*. In this way, we guarantee that calculated deltas for the target *sales* table must reside on the correct region server. Once the current regions get split due to load balancing and new regions are re-assigned to different servers, our partitioner gets notified and refreshes its local meta-data cache to keep track of this change. The CDC process together with the Partitioner component run continuously and keep *appending* correct source delta chunks to the delta input streams in the staging area of all the ETL flow copies running on region servers. The final deltas calculated by each local ETL flow instance are directly stored into local *sales* regions, which are further accessed through *scan* operations by analytic queries.

4.2.2 Consistency Model

In this subsection, we illustrate the consistency model that is supported by HBelt to deliver distributed snapshot maintenance to those distributed sub-requests issued by each individual query. An example is shown in Figure 4.3. A target HBase table is composed of three regions ($R1, R2, R3$) distributed across two region servers. At the upper left side, there is a traditional transaction log file recording eight transactions ($T_1 \sim T_8$) committed at the source side. The CDC process mentioned in the previous subsection is continuously extracting changes from the log file and sends

corresponding deltas[9] to the delta input streams in the staging area of both of the Kettle flow instances.

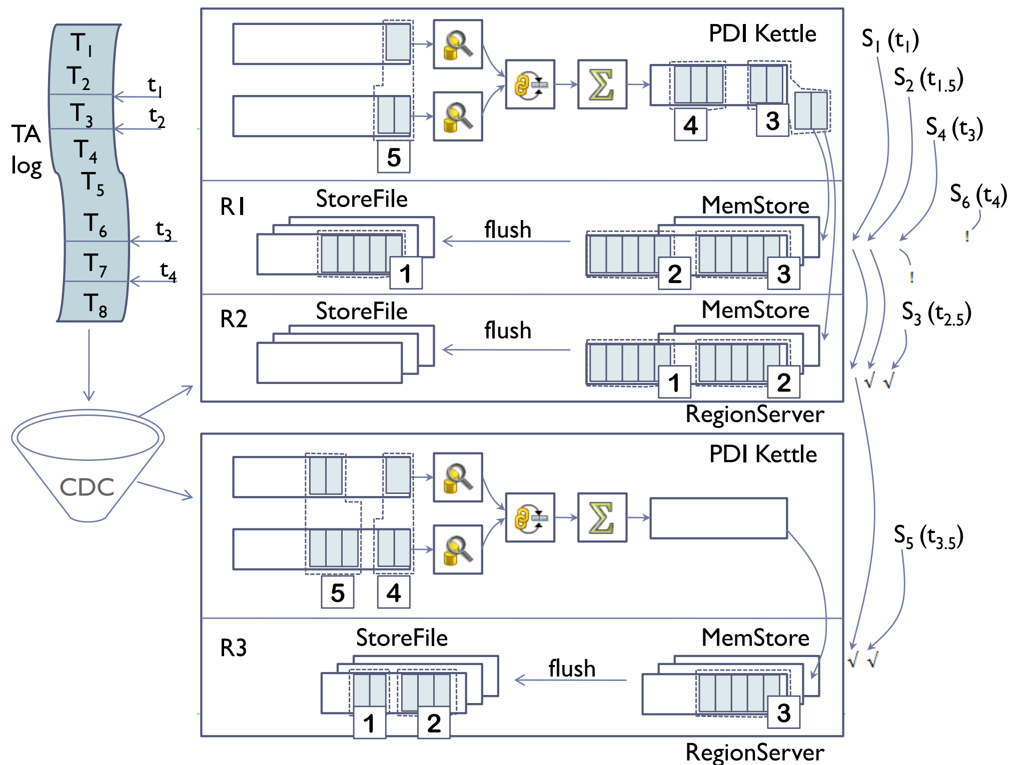


Figure 4.3: Consistency Model in Distributed Environment

Meanwhile, six queries have been issued to perform scan operations over regions stored in these two region servers at different points in time. The first query S_1 occurs at timestamp t_1 and scans all three regions, which immediately forces HBelt to refresh the HBase table with the change data (e.g. insertions, updates and deletions) that have been derived from a group of two source transactions T_1 and T_2 committed before t_1 and subsequently buffered in the staging area. As we can see from Figure 4.3[10], all these three local scan operations of S_1 have been successfully triggered each time the target updates (marked by the boxes with ID 1) calculated by the ETL flow instance become visible in the memStores (or

[9]Deltas are partitioned into smaller chunks depending on the arrival time of incoming queries and marked by corresponding ID (for example, deltas with ID 1 are related to t_1).

[10]Since scan is performed region-by-region in sequence according to sort order in HBase, the execution of S_1 is indicated by three remote procedural calls jumping from one to another.

in the storeFiles caused by flushing) of the region, respectively. Although the second query S_2 (which scans only region 1 and 2) is issued at a later time $t_{1.5}$, it still precedes the commit time of transaction T_3 (at t_2). Hence, it shares the same state of the HBase table as S_1 and has also completed. This matches our consistency definition that all queries have to see a consistent view of regions that are refreshed by the latest committed source updates preceding the query submission time.

The third query S_3 (scanning only region 2) occurs at time $t_{2.5}$, which succeeds the commit time of source transaction T_3 and precedes the commit time of a group of transactions T_{4-6} at t_3. This triggers HBelt to maintain the required region 2 (accessed by S_3) at the appropriate time t_2 only in the first region server with the updates extracted from T_3. Once those updates (marked by ID 2) are available in region 2, S_3 starts immediately. However, it is worth to note that region 1 gets refreshed also, since the local ETL flow instance is responsible not only for region 2 but for all regions (including region 1) in the same region server. Here we see that HBelt performs region maintenance lazily and only refreshes subsets of regions if necessary (i.e. region 3 in the second region server did not have to be maintained for S_3).

From t_3 to t_4, two queries S_4 (at t_3) and S_5 (at $t_{3.5}$) occur successively, which access regions 1 & 2 and only region 3, respectively. Both queries demand required regions to be refreshed by change data from source transactions committed before and at t_3 at the source side, i.e. T_{1-6}. Each copy of our ETL flow instance works independently for the respective query. In the first region server, S_4 has successfully scanned required content from region 1 while it waits for reading from region 2, since the expected tuples (marked by ID 3) have not arrived in the memStores of region 2 yet. However, S_5 can already proceed since the update it demands is already visible in region 3 due to completion of processing smaller source delta chunks. Meanwhile, the updates marked by ID 2 are also available in region 3 at this time. While these updates were lazily ignored by previous S_3, they are demanded to be processed by S_5. HBelt ensures that all regions are eventually consistent at a future point in time even with lazy region maintenance, since all source deltas are always available in the staging area even if they are not immediately processed.

The final query S_6 (arrives at t_4) also needs to scan the entire table, which demands region maintenance affected by T_7. Since neither of the Kettle flows has finished propagating these deltas to HBase, S_6 is suspended until the HBase table is refreshed with the correct deltas. In addition, we see that the source delta chunks belonging to source transaction T_8 have been eagerly pushed into the staging area in both region servers. Once

any future query occurs, these delta chunks will be immediately marked by an accumulated ID 5 and taken as input by those ETL instances whose locations are shared with the same regions accessed by that query.

In a nutshell, we apply a lazy maintenance scheme to all distributed regions. Regions that are demanded by incoming requests get maintained with consistent source deltas by separate ETL flow instances. Incremental ETL flows are triggered not only by query submission events but also specific to the access pattern of the regions that are required by the queries. Most importantly, all regions will eventually become consistent, which is driven by query arrival.

4.2.3 Distributed Snapshot Maintenance for Concurrent Scans

In this subsection, we show how ETL flows and distributed scan requests are scheduled in region servers to achieve the consistency model we introduced in the previous subsection. Recall that in HBase strong consistency is provided at the row-level and maintained in each region using a MVCC control instance (see Section 2.7). Therefore, for a query comprising a distributed scan request on a set of regions, there is no guarantee that tuples accessed across regions are strongly consistent, which breaks the consistency property we expect. To provide any incoming query with a consistent view of correctly maintained regions, we introduce a global state table (in master) and a number of local state tables (for regions in the region servers). The captured deltas that can affect multiple regions for a certain query are grouped into a single, global maintenance transaction (identified by a GID number). Those writes occurring in a single region are bracketed into a local transaction (identified by a LID number).

In the master node, there is a global state table maintained for bridging the gap between the source state and the state demanded by incoming analytical queries (see Figure 4.4) at the sink of the ETL flow, which is the target HBase table. The master node serves as a coordinator from which all source deltas and queries are directed to the target region servers. Initially, there is only one row entry (1, -, -) in the global state table, which indicates that all pending source deltas are assigned GID 1 and there is no arrival of any incoming query. The CDC process continuously keeps track of source changes and updates the *TA* column value with a group of source transactions (e.g. two committed transactions T_{1-2}) which are extracted from the log files. Their deltas are tagged with the same GID 1 and split to multiple chunks sent to all related region servers. The tuples from a delta

chunk that belong to the same global transaction and are buffered in the staging area of a single ETL flow instance are grouped into a so-called *delta batch*. Delta batches represent a group of source transactions using a single GID and are closed by the arrival of incoming queries. Directly after that, the first query S_1 (that scans all regions) occurs and makes its arrival visible in the *REQ* column. Later S_1 *memorizes* its GID number and gets directed to the first region server to perform its first scan operation on region 1.

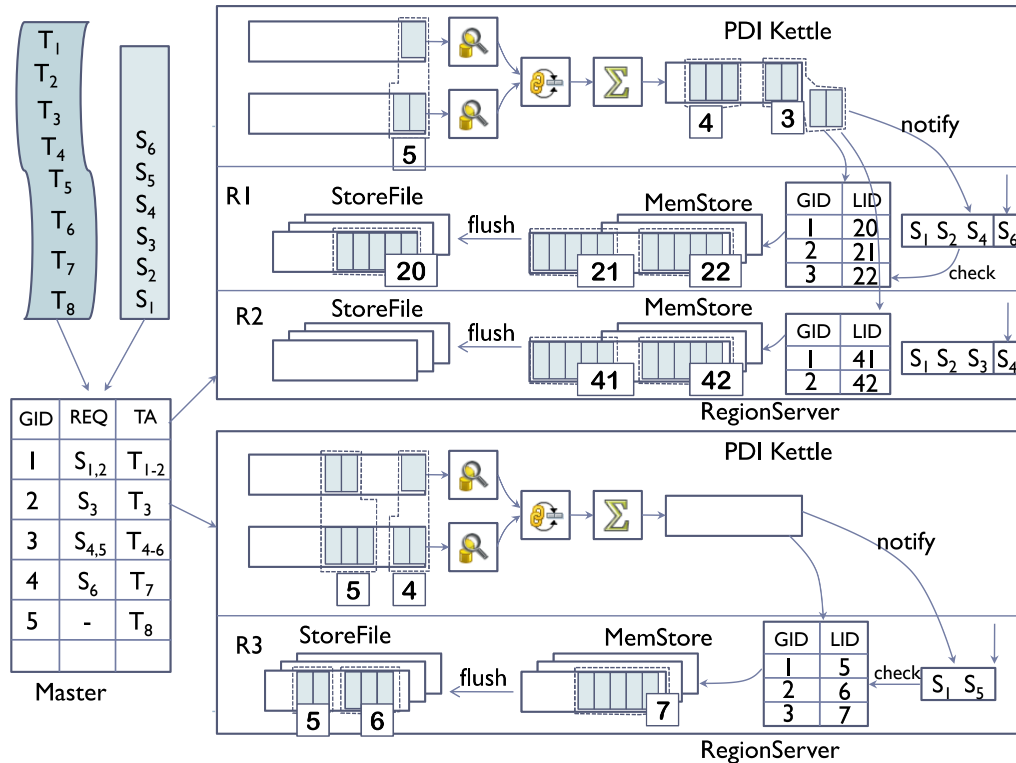


Figure 4.4: Distributed Snapshot Maintenance using Global/Local State Tables

In each region server, there are multiple local state tables, each of which is maintained by a local region to map the global transaction ID to its local one. Instead of directly accessing rows, S_1 first asks the local state table of region 1 whether there is an existing LID number assigned for its GID number (1). Recall that our incremental ETL flow instance maintains local regions lazily. With the request from S_1, the deltas (marked by GID 1) buffered in the staging area are taken as input for incremental ETL processing. Before the target deltas arrive in the memStores of region 1, the LID

value in the local state table is missing, thus blocking the execution of the scan operation of S_1 and pushing it into a request queue. Once any group of writes commit successfully in region 1, the local state table gets updated with a new entry (e.g. (1, 20)) which maps the GID number embedded in the input deltas to the LID number generated by the MVCC instance of that region. Each update of a local state table notifies all blocking scan requests to re-check the local state table for the new LID number. Once the LID number is found (i.e. LID 20), S_1 continues its scan operation on region 1 with this LID and its access is restricted to the tuples with LIDs less or equal to the given one. This protects S_1 from the phantom read anomaly which sees inconsistent deltas committed by succeeding global transactions (for example, deltas with GID 2 triggered by S_3), thus ensuring our consistency property in a single region.

After reading tuples from region 1, S_1 proceeds with its second scan request on region 2 in the same region server. This time region 2 has already been refreshed by the same local ETL flow instance and the local state table contains an existing LID (41) for the GID (1) of S_1. Therefore, S_1 starts immediately to read tuples from region 2 with LID 41. The same happens when S_1 jumps to the second region server for region 3. Although all regions maintain their own MVCC instances for independent concurrency control, the scan requests from the same query are still allowed to achieve a consistent view from all these three regions based on the consistency model we built. Finally, the given example ends with a state where S_6, which accesses region 1, is suspended due to its missing target deltas (with GID 4) and the same happens when S_4 gets pushed and blocked in the request queue of region 2. Note that, the last entry of the global state table (5, -, T_8) indicates that our CDC process is still capturing deltas and sending split delta chunks to separate region servers with the same GID number 5. With an arrival of any query, *REQ* column will get filled and that query will be related to the input deltas with GID 5. In this way, we achieve our distributed snapshot maintenance for concurrent scans in HBelt.

4.3 Experiments

The objective of HBelt is to provide scan operations in HBase with real-time data access to the latest version of HBase's tables by tightly integrating an ETL engine, i.e. Kettle, with HBase. Though current Kettle (since Version 5.1) has implemented "HBase Output" step towards Big Data Integration, in our scenario, sequential execution of a single Kettle flow at

once to maintain target HBase tables for time-critical analytics could lead to long data maintenance delay at high request rate. In this section, we show the advantages of our HBelt system by comparing its performance with the sequential execution mode in terms of maintenance latency and request throughput. We mainly examine the performance improvements by using data partitioning techniques in HBelt.

In the experiments, HBelt ran on a 6-node cluster where are a node (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM, 1TB SATA-II disk) served as the master and the remaining five nodes (2 Quad-Core Intel Xeon Processor X3440, 4×2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) were the slave nodes running HBase region servers and extended Kettle threads (see Subsection 4.2.1). Meanwhile, the same cluster was used to accommodate an original version (0.94.4) of HBase connected with a Kettle engine (Version 5.1) running on a client node (Intel Core i7-4600U Processor, 2×2.10 GHz, 12GB RAM, 500GB SATA-II disk) to simulate the sequential execution mode.

We used TPC-DS benchmark in our test. A *store_sales* table (with SF 10) resided in HBase and was maintained by a Kettle flow with the update files *purchases* (♯: 10K) and *lineitems* (♯: 100K) generated by TPC-DS *dsdgen*. The maintenance flow is depicted in Figure 4.5. Purchases and lin-

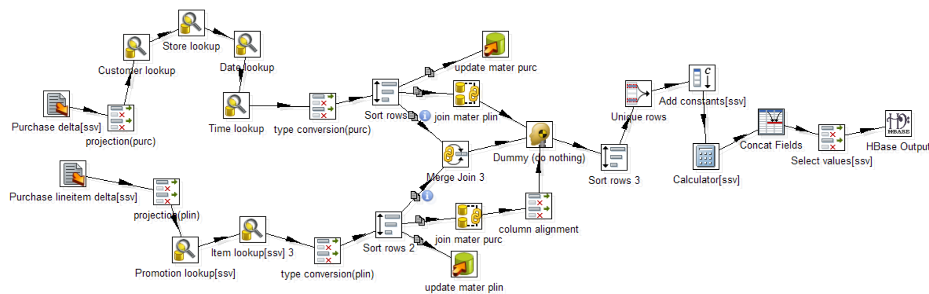


Figure 4.5: Test Maintenance Flow in Kettle

eitems are the delta files and are joined together in an incremental fashion after applying several surrogate key lookup steps. The intermediate join results are further aggregated as the final delta rows for the target store sales table. In sequential execution mode, the source delta files (purchases & lineitems) resided in the client node and were used as input for the Kettle flow to populate the store sales table in the 6-node HBase cluster using HBase Output. However, in HBelt mode, these source delta files were initially stored in the master node and later continuously distributed and fed to the five slave nodes where two input rowsets were used to buffer delta rows as delta input streams (instead of CSV Input steps).

Furthermore, in contrast to sequential execution mode, each region was the target output instead of "HBase Output" step.

Firstly, the *store sales* tables were evenly pre-split to 10 regions with non-overlapping row key ranges over 5 region servers, thus each region server was active and managed 2 regions. Secondly, the request load consisted of a thousand scan operations in which each individual Region[1→10] was scanned by 50 scan operations, subsequent 100 operations scanned Regions (1∼3), 100 operations scanned Regions (4∼6), 100 operations scanned Regions (6∼8), 100 operations scanned Regions (8∼10) and the rest 100 operations scanned the entire table. Hence, each request required in average only 2/7 of the table to become up-to-date before it was executed. Finally, we generated a set of delta files *purchases* and *lineitems* of ten sizes {♯: (10K & 120K), (20K & 240K), ..., (100K & 1200K)} each of which was further split to 1000 chunks to simulate the delta inputs for the 1000 scan requests. In each chunk only a 2/7 portion in average is needed to refresh the necessary regions for one request.

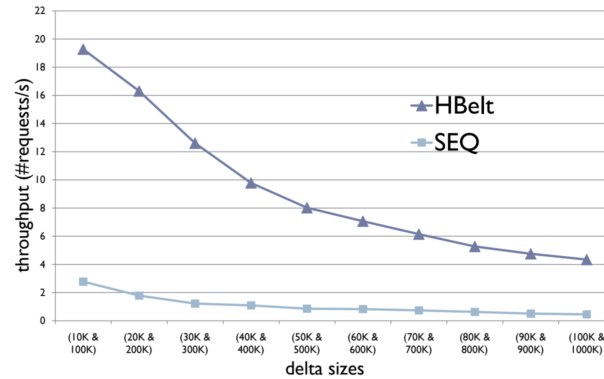


Figure 4.6: Request Throughput after Issuing 1000 Requests using Diverse Delta Sizes

The request throughputs with different delta size settings are shown in Figure 4.6. As the baseline, the request throughput in SEQ decreases steadily from 2.78 (♯requests/s) to 0.46 (♯requests/s) with increasing delta sizes, which indicates growing maintenance overhead. The throughput in SEQ mode is much lower than that in HBelt since two scan operations have to be executed sequentially no matter how many deltas are really needed to answer certain request. HBelt provides much higher throughput (19.28 to 4.35 ♯requests/s). The efficiency is two fold. Due to data partitioning, HBelt is able to propagate deltas for concurrent requests with non-overlapping key ranges at the same time. For example, a scan operation which accesses Region(1∼3) has no conflict with another scan opera-

tion which touches Region(4∼6). Separate ETL pipeline can refresh independent regions at the same time. Meanwhile, since deltas were split and distributed over multiple ETL pipeline instances, the size of input deltas dropped drastically and the latency became less as well. In addition to data partitioning, pipelined Kettle still provides data pipelining parallelism for multiple concurrent requests arriving at the same region server.

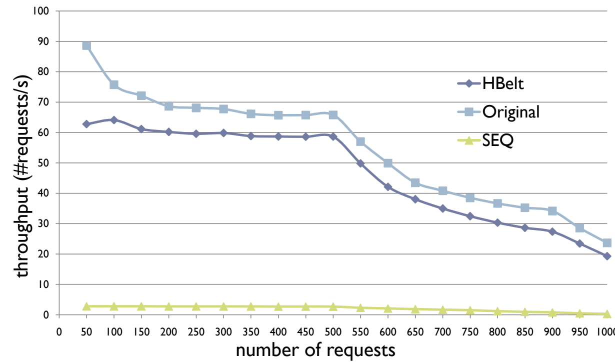


Figure 4.7: Request Throughput with Small Deltas (10K purchases & 100K lineitems)

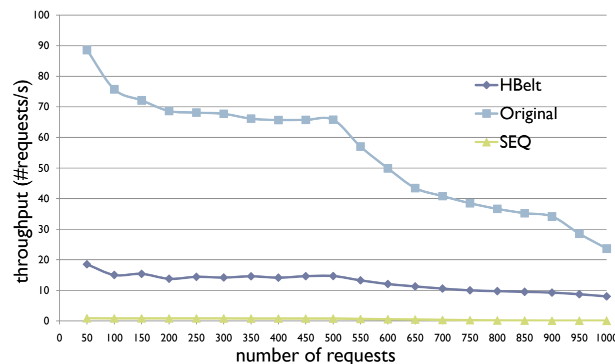


Figure 4.8: Request Throughput with Large Deltas (50K purchases & 500K lineitems)

Figure 5.4 & 4.8 compare the throughput with increasing requests among three settings: HBelt, sequential execution mode and an original HBase setting which does not have maintenance overhead incurred by our ETL pipelines. With small delta sizes (10K purchases & 100K lineitems), HBelt achieves performance much similar to the original HBase which does not guarantee data freshness. However, as the size of the delta grows, the request throughput of HBelt dropped significantly while it

still outperforms the sequential execution mode.

4.4 Summary

We first described our HBelt system for distributed snapshot consistency using a partitioned incremental ETL pipeline. HBelt exploits the essence of the cluster processing architecture of both HBase and Kettle systems. Data partitioning parallelism is achieved by running a redundant ETL flow instance on each region server node. A range-based partitioner is implemented by a customized Kettle partitioner with HBase-specific meta-data. Each distributed redundant ETL flow instance is further optimized with data pipelining parallelism to increase the throughput of delta batch processing. Both data partitioning and data pipelining parallelism increase the performance of ETL job execution.

By grouping source deltas and executing them in a global transaction manner, we built our consistency model which guarantees a consistent view of HBase regions at a global level. Furthermore, we defined the freshness of each region as the latest version preceding the request submission time. The freshness and consistency for a set of regions are maintained by the global and local state tables, which results in distributed snapshot consistency.

The experimental results show that HBelt is able to reduce maintenance overhead and increase request throughput for real-time analytics in HBase. We believe that the implementation of HBelt is conceptually applicable to any scalable data stores for time-critical decision support queries using distributed snapshot maintenance.

5 Scaling Incremental ETL Pipeline using Wide-Column NoSQL Databases

In Chapter 4, we introduced our HBelt system which uses a distributed streaming pipeline to refresh data warehouses on demand. In this chapter, we address classic streaming processing problems (e.g., back pressure, slow bottleneck operators, elasticity, etc.) and propose our solutions, i.e., Elastic Queue Middleware (EQM) and HBaqueue. HBaqueue replaces memory-based data exchange queues with scalable distributed stores (e.g., HBase [Geo11]) in stream processing systems. EQM integrates HBaqueue to enable slow bottleneck operators to auto-scale with upstream queue-shards in HBaqueue [QD17c].

The remainder of this chapter is as follows. We illustrate the motivation of achieving elasticity for distributed streaming processing systems (especially for HBelt) in Section 5.1. The novel idea of "Elastic Queue Middleware" which uses HBase as the streaming queues are described in Section 5.2. A prototype of EQM (i.e., HBaqueue [Gre17]) was constructed based on this proposal and examined with experimental results in Section 5.3.

5.1 Motivation

With high demand for real-time business intelligence, a new generation of distributed data stream processing systems has been developed to address the *velocity* property from the 4 V's of big data. Scalability is achieved by running streaming processing jobs in a distributed cluster environment. Unbounded data streams are split into multiple contiguous small, bounded batches that are executed concurrently by parallel instances of different operators, which forms a *distributed streaming pipeline*.

HBelt is one such example and other classic examples are listed as follows. Spark Streaming [ZDL+13] proposed discretized streams (*D-Streams*) which are internally cut into successive batches, each of which is a resilient distributed dataset (*RDD*, as storage abstraction) and can be

executed in their underlying pull-based, batch processing engine. Flink [CKE+15] follows the dataflow processing model and uses *intermediate data streams* as the core data exchange abstraction between operators. The real data exchange is implemented as the exchange of buffers (similar to batches) of configurable sizes, which covers both streaming and batch processing applications. Additional work like Storm+Trident [tri] and S-Store [MTZ+15] addresses transactional consistency properties (exactly-once processing and ordered execution properties during concurrent access to shared state by parallel operator instances) and also fault-tolerance, at batch-wise scale.

Recent work [SAG+09, GSHW13, CFMKP13, WT15] has also contributed to achieving elastic processing capability in response to workload fluctuation in distributed streaming processing systems. On detecting spikes in data rates, bottleneck operators are scaled out to parallel operator instances across cluster nodes to keep stable throughput. After the workload spikes, over-provisioned resources are further released for better resource utilization. One use case is the Internet of Things (IoT) applications, for example, smart cars. Sensors embedded in automobiles continuously send event streams to the cloud where real-time streaming analytics is performed. Given varying input data volume, elasticity is important for the streaming pipeline to scale out at peak hours and later scale in at off-peak times. Another use case is real-time data warehousing which relies on streaming ETL [MAZ+17] engines to refresh warehouse tables. With continuous updates occurring at the data source side and analytics requests (with different freshness and deadline needs) issued to the data warehouses, streaming engines need to process different amounts of input data in time windows of various sizes in an elastic manner.

With drastically increasing data rates, the streaming buffers between processing operators fill up fast, which normally results in back pressure. This back pressure impact would not be resolved until the system first detects where the bottleneck is, determines the scale-out degree, lets the deployment manager apply resources, spawns threads and finally reroutes dataflow. Especially for stateful operators, additional state-shuffling cost is mandatory. To alleviate back pressure, streaming buffers can spill to disk and also scale across cluster nodes which most of the full-fledged scalable data stores have already implemented. While most of the elastic streaming processing engines first scale the operators and then set the buffers, we argue that the buffers that really *hold* the overflowed tuples should first scale and then set the operators appropriately. Instead of re-inventing scalable buffer from scratch, we address the “operator-after-buffer-scale” logic with an Elastic Queue Middleware (EQM) which utili-

zes existing scalable data stores to implement a scalable buffer for elastic streaming processing.

5.2 Elastic Queue Middleware

In this section, we describe how to embed our Elastic Queue Middleware (EQM) in a general streaming processing engine through a set of elastic queue primitives. Furthermore, we provide details in implementating EQM primitive interfaces using HBase [Geo11].

5.2.1 Role of EQM in Elastic Streaming Processing Engines

Existing streaming engines contain elasticity components to adapt to changing workloads at runtime. These components are shown in the upper part of Figure 5.1. In order to observe workload fluctuation, a *monitor* collects processing metrics from per-operator executions in a streaming pipeline and sends these metrics periodically to a *scale handler*. The metrics usually cover throughput (processed tuples per second), congestion (blocking time of tuples in the streams between operators) and resource consumption (CPU cycles, memory).

To alleviate stream bottlenecks in time, the scale handler relies on (control) algorithms [SAG+09, GSHW13] to detect bottleneck operators in a streaming pipeline (given spikes/decline in data rates) and to make accurate decisions on the degree of parallelism for scale out/in. The result of the control algorithm is sent to a *deployment manager* which balances parallel computation at runtime. On receiving a scale-out request, the deployment manager immediately applies for resources in specific nodes. With resources granted, it spawns new threads to generate replicas of the bottleneck operator and further allocates memory for data exchange buffers between new replicas and their upstream operators.

If the bottlenecks are stateless operators, after scale out/in, their upstream operators simply re-distribute (e.g. round-robin) output tuples evenly across new operator replicas. For stateful operators, a *state management* component is introduced to manage their internal processing states and scale out/in the state for new operator replicas. Most importantly, it has to guarantee the state consistency during scaling. The processing state is generally maintained in key-value data structures through explicit state management APIs. For scale out/in, one can pre-partition the state on the key at deployment time and only shuffle the state partitions

across a varying number of operator replicas at runtime [WT15]. For better load balancing on skewed workload, another solution is to re-partition the local state using different partition functions at runtime before shuffling [GSHW13, CFMKP13]. In both cases, the upstream operators have to know the new locations of the shuffled partitions (i.e. new partition schema) in order to reroute the dataflow. This information may be acquired from a dedicated *partitioner*. Moreover, to guarantee fault tolerance, the processing state is periodically checkpointed and restored in new nodes by a *failure handler* in case of failure.

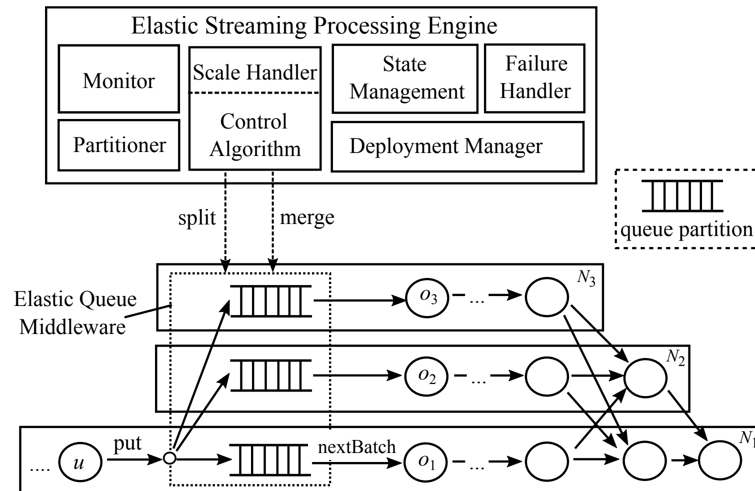


Figure 5.1: Embedding Elastic Queue Middleware in Streaming Processing Engine

Since the state migration (re-shuffling) happens on the fly, one has to prevent re-shuffled state partitions from being processed by inconsistent stream tuples before successful dataflow rerouting. One simple solution is to block the upstream operators and prevent them from sending output tuples while the state is being (probably repartitioned and) re-shuffled across cluster nodes. This results in back pressure, especially when the input data rates increase drastically. Another general solution is to buffer the output tuples sent from upstream operators temporarily until state migration finishes. Assuming that the workload spikes are long-time, not transient, buffers may be overloaded and frequently spill to disk. If upstream backup [CFMKP13] is used, a scale out of upstream operators can be caused in a cascade fashion. Alternatively, the buffers can be split along with the processing state and placed in the same nodes where new partitioned operators (replicas of original bottleneck operator) are loca-

ted, which would not overload upstream operators. We call this locality-affinity buffering.

Instead of *actively* detecting and replicating bottleneck operators, we let the operators scale along with an elastic buffer (more concisely, a *queue*[11]) in a *passive* manner. At the bottom part of Figure 5.1, an example is shown. An operator u was initially only attached to a downstream operator o_1 through an elastic queue with one queue partition having a pre-defined maximum size as threshold. On a spike in data rates, o_1 becomes the bottleneck and the queue gradually scales out to three partitions in nodes N_{1-3} to balance the load. The deployment manager spawns new threads o_2 and o_3 in the nodes N_2 and N_3, respectively, to cope with the workload spike. To eliminate the overhead of thread construction at runtime, the threads can be created, suspended at deployment and be woken up at runtime when the queue partitions are *spread* on the same node. Moreover, to increase parallelism, the operators that can be partitioned on the same key are grouped and replicated together across cluster nodes until a new partition key is used in downstream operators, which requires a shuffle phase.

In the *era* of big data, a wide spectrum of scalable data stores [Cat11] has been developed and extensively used in industry. The four main types of NoSQL databases are: key-value, wide-column, graph and document databases while wide-column databases (e.g., HBase) are characterized by outstanding write and sequential read performance, as well as excellent horizontal scalability and rich extensibility.

Instead of writing a new, internal, elastic queue from scratch, we propose EQM as the solution which wraps scalable data stores and exposes only a basic set of methods to the existing components of an elastic streaming engine. These EQM primitives are *put, nextBatch, split* and *merge*. In the following, we explain how a scalable data store, i.e., HBase, can be wrapped as EQM and how the four EQM methods are implemented.

5.2.2 Implementing EQM using HBase

HBase shows high scalability and low data-access latency. In Section 2.7, a detailed introduction to HBase is given. Sequence ids are stored and maintained in both store files and in-memory indices to guarantee scan consistency and performance.

As compared to original HBase, in this subsection, we describe how

[11] Instead of buffer, we want to achieve "write once; read once"queue feature with *enqueue* and *dequeue* operation support in this context.

we modify the internal core of HBase to implement the four primitive methods of EQM: *put*, *next-Batch*, *split* and *merge*, as depicted in the elastic streaming processing diagram in Figure 5.1.

Put. Since most of the streaming processing engines attach batch ids to stream tuples to distinguish different batches at runtime, we assume that the incoming key-value (*kv*) tuples contain additional batch id (*bid*) information and the original add method is extended to *add*(*kv* | *bid*). We modified HBase to replace the original sequence-id list with a new *batchId-seqId map* which maps a (streaming-level) batch id to multiple (HBase-specific) sequence ids. Depending on the upstream operators, all the tuples belonging to one batch can be added tuple-by-tuple (each requires an add method invocation), all together (just one method call) or as multiple subsets. All sequence ids assigned to the adds from one batch are grouped together as the values of that batch-id key which represent a contiguous subset of the entire sequence-id list. The example in Figure 5.2 shows that an in-progress add method is called to append a *kv* tuple to the memstore and meanwhile to put its assigned sequence id (8) in the sequence list associated with the batch-id key b_4.

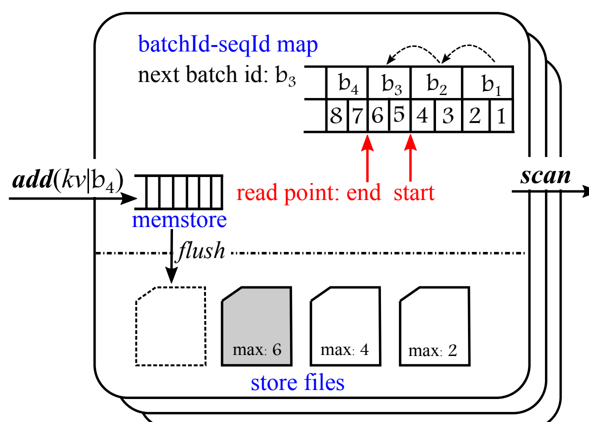


Figure 5.2: EQM-specific Partitions in HBase

NextBatch. Similar to the read point used for the original sequence-id list, three new variables are introduced for the *nextBatch* method: *next_batch_id*, start read point (*startPt*) and end read point (*endPt*). Moreover, a new *DequeueScanner* is implemented to scan the memstore and store files in a queue partition. When a dequeueScanner is instantiated, it tries to fetch the next available batch id in the *batchId-sequenceId* map, as the map is sorted on the increasing batch id. If any add operation containing sequence id associated with the fetched batch id has not commit-

ted yet, the dequeueScanner blocks the current calling thread, which also suspends the downstream operator thread, until all adds of this batch have committed. The dequeueScanner uses the fetched batch id to find its sequence-id list in the map and assign the minimum and maximum sequence ids in the list to the startPt and endPt, respectively. Memstore and store-file scanners are modified so that they only return the tuples that have their sequence ids falling in the range of [startPt, endPt]. It is important to note that a store file contains a max sequence id which indicates the greatest sequence id among all its key-value tuples. Given a [startPt, endPt] range and a list of store files sorted on their max sequence ids, only required store files are scanned, thus significantly reducing disk I/Os. In Figure 5.2, invoking the *nextBatch* creates a memstore scanner and multiple store-file scanners with [5, 6] as the read point range after looking up the map with b_3 as the key. Only the store file with max sequence id (6) is hit for scanning. A subsequent nextBatch call would stall until the $add(kv \mid b_4)$ has its sequence id as (8) commits.

Split. HBase supports automatic splitting and manual splitting. One can set the maximal allowed size of a store file at deployment time and let HBase automatically split those queue partitions that have one of its store files exceeding this threshold at runtime. Automatic splitting has the main advantage that there is no need to implement extra elasticity control algorithms and monitoring component in streaming engines. However, the drawback is that the decision of scaling out/in is limited to the HBase-internal metrics (e.g. max file size), thus losing the flexibility of tuning scaling using different metrics, e.g. processing throughput, resource consumption, etc. In addition, if the max file size is set too high, EQM would not react to workload variations immediately and not scale out in time. If the max file size is set too low, EQM would be load-sensitive and be busy with splitting, thus incurring lots of unnecessary network I/Os. All of these drawbacks can be compensated by letting the elasticity control algorithms invoke HBase-internal *split* methods to manually split partitions on given split keys, whereas new overhead is incurred that the streaming engines need to keep track of load distributions across partitions. The choice of two splitting scheme is a trade-off between software engineering costs and performance.

Merge. We see that the nextBatch method does not immediately remove the scanned tuples from the storage, although the performance is not impacted as the dequeueScanner selectively scans store files with given read point range. If the workload shows a decline, we can merge queue partitions to less number of partitions by calling the merge method in HBase. Before that, a local compaction is processed in each partition to re-

duce the amount of store files transferred through the network. As mentioned in Subsection 2.7.2, the working principle of the compaction process is to first read out tuples from the store files and write them to a single one. Hence, another new *QueueCompactor* is implemented. It fetches the minimum sequence id from the sequence-id list of the most recently dequeued batch id and skip unnecessary store files having smaller maximum sequence id. Such store files have been scanned and are not needed any more. Original version of memstore and store-file scanners are then used to finish the rest of the compaction and generate a much smaller store file. In the example, the store files with max sequence id (4) and (2) are skipped during compaction. However, HBase does not support automatic merging. The right time of scale in thus has to be determined by the control algorithm of elastic streaming engines.

5.3 EQM Prototype: HBaqueue

In previous section, we showed how to modify the internal core of HBase to simulate a distributed queue middleware. However, several other critical problems still exist since HBase was not originally designed as a *queue*, where data is enqueued (put) once and dequeued (scan) once, rather a general data storage which is optimized for write-once-read-many-times scenario. For example, HBase relies on compactions to reduce overheads for reading the same tuples multiple times while for a queue, each element is intended to be dequeued only once. Moreover, the load balancing logic in HBase would not consider queue-specific characteristics.

Therefore, in addition to modifying read and write paths in HBase, HBaqueue [Gre17] is proposed to address further relevant aspects and provides experimental comparisons with original HBase. HBaqueue is mainly designed for our on-demand ETL scenario, thus guaranteeing that batches of varying sizes would be dequeued in a strict order according to the monotonically increasing batch ids. Besides, per batch tuple-dequeueing is evenly distributed across queue partitions without hotspots. In the following, we present several key modifications made to re-salting, splitting, load-balancing and compaction followed by dequeue and enqueue performance improvements.

5.3.1 Compaction

As introduced in Subsection 2.7.2, a compaction process introduces additional I/O operations as small store files would be read from and writ-

ten back onto persistent storage. During compaction, bloom filters and store-file indexes would be created as well, which also introduces CPU overhead. Such negative impact on performance is especially significant when HBase is used as a queue since the data that is intended to be read only once is read and rewritten several times.

To measure the impact of compactions on performance, a test was given which compares the default HBase setting and a setting with compaction disabled in terms of enqueue (put) latency. The test workload consisted of 210 batches, each of which contains 100K records, i.e., overall 21 million key-value pairs.

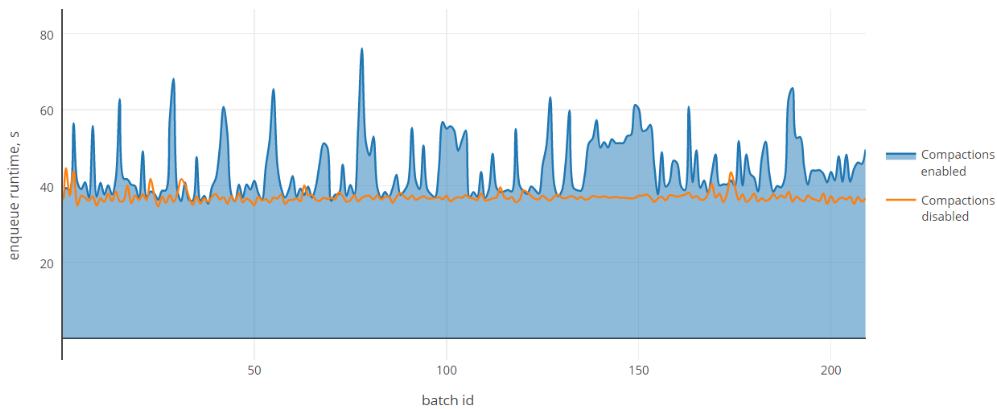


Figure 5.3: Enqueue Latency with and without Compactions in HBase

As shown in Figure 5.3, in case when compaction was disabled, the elapsed time of enqueueing all the input batches was 20% lower than that in the case when compaction was enabled (i.e., 130 minutes vs. 156 minutes). All of the 70 compactions occupied 72 minutes which is 46% of the total test runtime. The test showed that compactions significantly increase the number of I/O operations and result in 20% overall write performance loss. Therefore, it is not recommended to compact small store files into bigger ones at runtime, since each element would be dequeued only once. Meanwhile, as described in Subsection 5.2.2, a so-called *QueueCompactor* is introduced to periodically garbage-collect dequeued store files, which removes additional I/O costs incurred by original compactions.

5.3.2 Resalting

To avoid hotspots, row keys need to be salted (e.g., with random or hash prefixes). Since random salting is easy to implement and scales well, be-

sides, it allows records of a single batch to be dequeued in an arbitrary order as well, a random function is a better candidate for salting.

However, random salting has negative impact on read performance, because records of the same batch are not stored adjacently. A single row key range is insufficient to retrieve records of a specific batch, so it needs to scan the entire key range, extract the batch id from the row keys of each record and compare them to the searched one. The situation gets worse if there are several store files in the Region, since each lookup will be performed over all of them.

A workaround to introduce the random salting to HBase without significant modifications to its source code is to modify the salt of a rowkey on a *RegionServer* before writing it to the store in such a way that all row keys in a region have *the same salt values*. The new value of a salt can be retrieved from either the start or the end key from the key range of a region, so that the new row key will still lay in the key range of a region. Thus, all records of a region will be stored in the same order as if their row keys would contain no salted prefix. Row key modification is performed in the prePut() callback method of a *RegionObserver* coprocessor.

5.3.3 Split Policy

As mentioned in Subsection 2.7.3, the current split point determination algorithm in HBase uses the median row key of all records in the largest store file as the split point. By applying the new resalting function, all row keys of a region are salted with the same value, hence the split point will also be salted with the same value. After splitting, one daughter region has key range [start, split) and the other has key range [split, end). Assume that split is selected as the same value as the original start key. The first daughter region is empty and the second daughter region still receives the entire load, which would not take any splitting effect. The original load balancing would apply only if a new split-point determination algorithm is introduced. This algorithm should select a split point which value is different from the current salt value of the parent region.

Since the random salting ensures even load distribution over the entire row-key space, to achieve equal read & write distribution among daughter regions after split, the salt value of a split point should be equally distant from the salt values of the original start and end keys of the parent region. Hence, the split point is calculated as the median value between the start and end keys. A new split policy class is provided and the new split-point function (called *getSplitPoint()*) can be invoked in the *RegionS-*

erver to calculate the split point based on our custom rule.

In addition, the original split policy also requires modification together with the new split-point function. The original split policy decides to split a region if the size of its largest file exceeds a certain threshold (called split size). This algorithm relies on an assumption that each of the two daughter regions has roughly half of the data of the parent region. However, as all rows in a region have the same salt value while *getSplitPoint()* selects the median value from the original key range, splitting would move all data of the parent region to one of the daughter regions having key range [start, median). It is undesirable that the split size still stays the same as before and this new daughter region becomes eligible to split again. The same cause leaves the content of the other daughter region empty. Since batches are enqueued and dequeued in sequence, the empty daughter region would not serve reads until all previous pending batches in its sibling daughter region are dequeued. Meanwhile, the empty region may also grow fast to the split size as new incoming tuples can not be dequeued until its sibling daughter has finished the previous batches.

If any of the daughter regions starts splitting before the read load is distributed between two daughter regions, the decision made on the necessity of the split is *premature*: new splits would have been suspended until read load starts to dequeue tuples, which shrinks the region size.

To avoid such premature splits, new split policy should consider only the growth of the store size right after the read load is distributed evenly over new daughter regions. As is proposed in Subsection 5.2.2, an internal per store *next_batch_id* is maintained to indicate the most recent batch id that is used by the current read load to dequeue tuples. The growth of store files could thereby be calculated over those store files with the sequence ids which are greater than the minimum sequence id of the current *next_batch_id*. In this way, new split policy considers only the increase in the store size instead of calculating the entire region size as set in original HBase setting.

5.3.4 Load balancing

As introduced in previous part, by applying new split policy over resalted region tuples, a parent region is split into two daughter regions, one of which is empty. A new load balancer is introduced which selects the empty daughter region as the candidate to be moved to a remote *RegionServer*. The benefits are two-fold. On one hand, as this daughter is empty, it can be freely moved to the least loaded with very tiny network over-

head while on the other hand, data locality is ensured once subsequent read load starts to dequeue tuples from this region on the same node. The rebalancing could be triggered right after splitting by invoking the *postSplit()* callback method of the *RegionObserver* coprocessor.

Apart from load-balancing after splitting, the new load balancer uses a different candidate function to predict a *RegionServer*'s load which is the sum of differences between batch ids of the start and end keys of all regions hosted by this *RegionServer*. This load metric represents the amount of pending batches buffered on a certain *RegionServer*. Regions with high load could then be migrated to the least loaded servers, which increases the total dequeue-throughput.

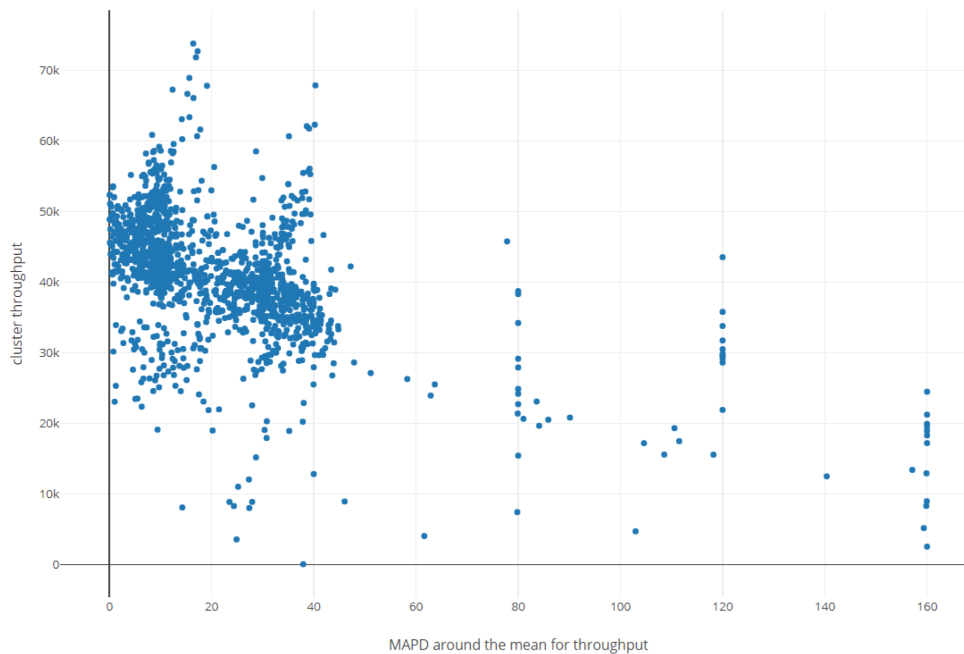


Figure 5.4: Correlation between Cluster Throughput and MAPD around the mean for Throughput

New Load-balancing Metric. With random values set as the salt prefixes, the tuples of each batch are evenly distributed among the entire key ranges. Hence, all nodes of an ideally balanced cluster should have the same throughput which is measured as the number of processed client request per second. The quality of the load distribution is measured as the quality of the throughputs distributed across all cluster nodes. The *mean absolute percentage deviation* (MAPD) around the mean for throughput is used to quantify the load-balancing quality. The formula is listed as

follows:

$$M = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{A_i - \overline{A}}{\overline{A}} \right|$$

In this formula, A_i denotes the throughput of the i-th node while n is the number of nodes in the cluster. $\overline{A}$ represents the mean throughput value in the cluster (i.e., the throughput of the global cluster divided by n).

It is important to note that the lower the MAPD is, the better is the load-balancing quality and the higher is the cluster's throughput. The assumption has been verified by several tests, as shown in Figure 5.4. The tests were carried out on HBaqueue and a clear correlation between cluster throughput and MAPD has been found, which empirically ground the usage of this metric.

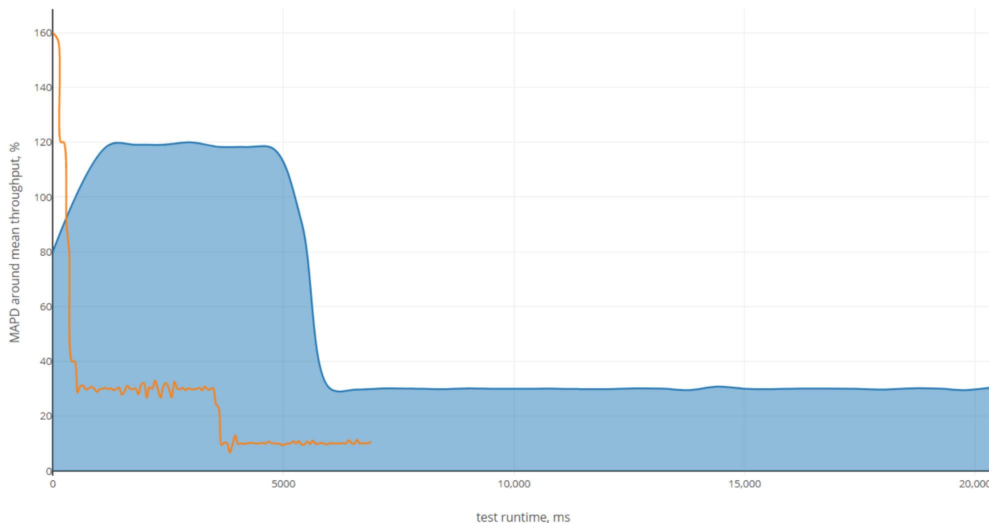


Figure 5.5: Load-balancing Quality in HBaqueue and HBase

Load-balancing Quality Comparison. In Figure 5.5, tests were conducted to calculate the MAPDs around the mean for throughputs in Hbaqueue and HBase. The comparison of the average throughputs across *RegionServer* nodes shows that HBaqueue's load balancer (orange line) performed 3x better than HBase's load balancer (blue line) in average. With better load-balancing model, HBaqueue was able to distribute read/dequeue load more evenly across nodes, thus accomplishing the entire batches within 3x shorter time as compared to HBase.

Load Distribution Comparison. Figure 5.6 compares the real load distribution in HBase (left) and HBaqueue (right) at runtime in a 5-node cluster. Plots traced from each node represent the changing load factor on each *RegionServer*. The load factor was calculated as the current throughput in requests per second divided by the entire throughput of the global cluster. The load distribution plots show that HBaqueue started to use all servers only after 10 minutes while it took 1 and half hours for HBase to get the load balanced.

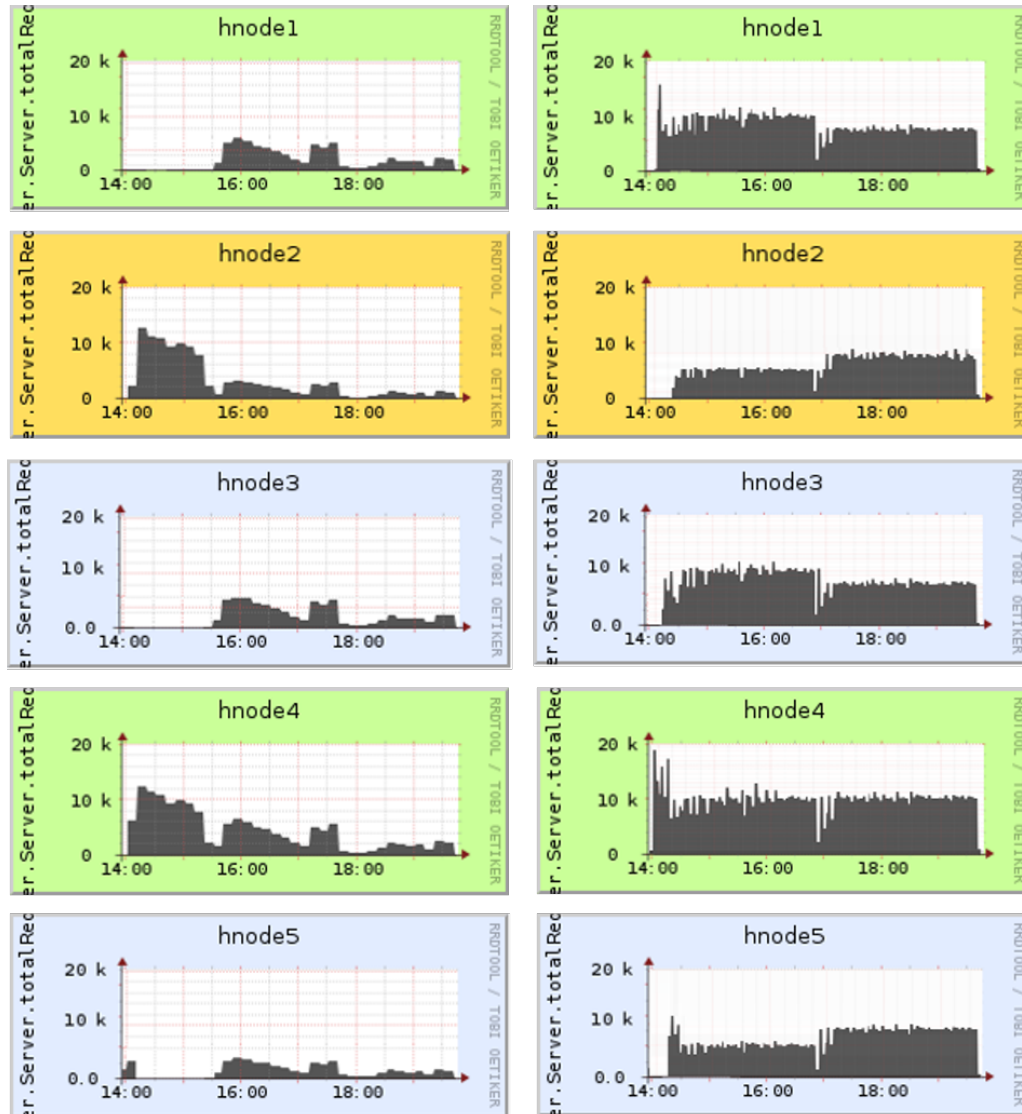


Figure 5.6: Load Distribution in HBase and HBaqueue

5.3.5 Enqueue & Dequeue Performance

Enqueue. By using new load-balancing scheme and disabling compaction, HBaqueue outperforms default HBase at logarithmic scale (see Figure 5.7). The average batch-enqueueing (push) time in HBaqueue is 14% better than that in HBase: 3258 ms vs. 2793 ms.

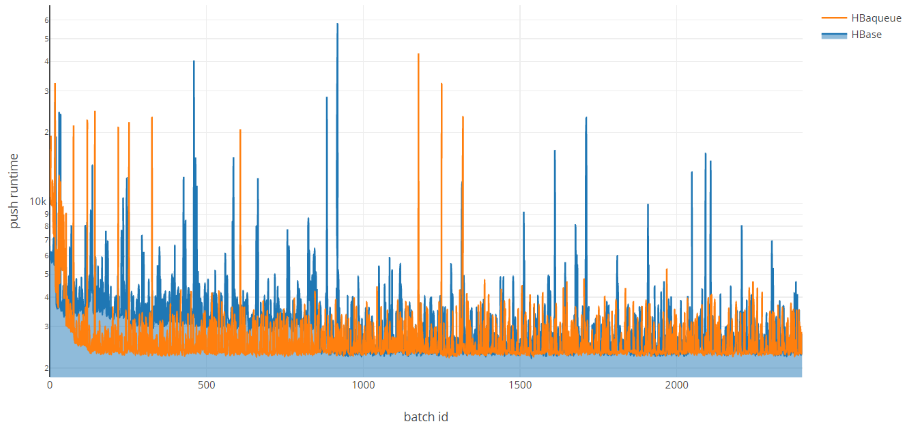


Figure 5.7: Batch-Enqueueing Latency in HBaqueue and HBase

Figure 5.8 depicts the distribution of latency results also at logarithmic scale. The median enqueue-latency of HBaqueue is 2368 ms which is 17% lower than that of HBase. Additionally, it shows that HBaqueue performance was more stable since there are smaller number of outliers of execution latencies in HBaqueue.

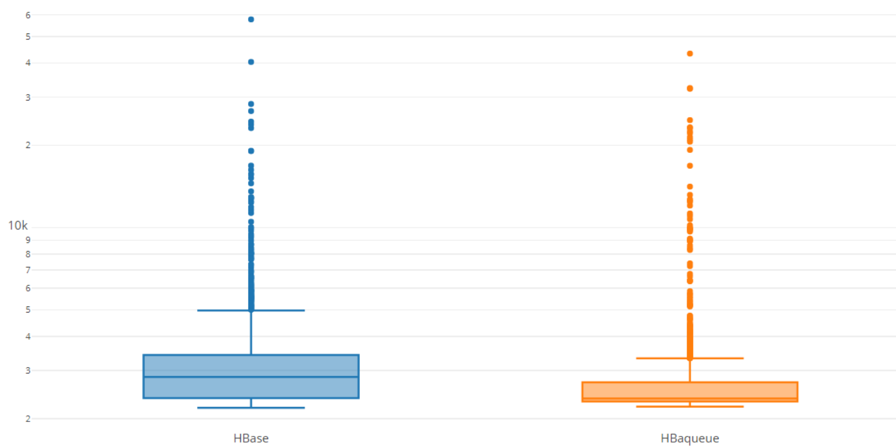


Figure 5.8: Latency Distribution of Batch-Enqueueing

Dequeue. As shown in Figure 5.9, the read/dequeue performance is drastically improved at logarithmic scale in HBaqueue thanks to the improved load-balancing model and the modified read path. Dequeue latency doesn't depend on the storage size because only the store files that contain the matching batch id are filtered and accessed in the read path. Furthermore, HBaqueue also achieves benefits of block cache, since less store files are scanned during the dequeue processing, hence it is more likely that subsequent batches can be found in block cache. Moreover, with growing table size, read performance decreases significantly in HBase, since the rows of a batch contain different salt values in a region and all store files have to be scanned to answer a dequeue request.

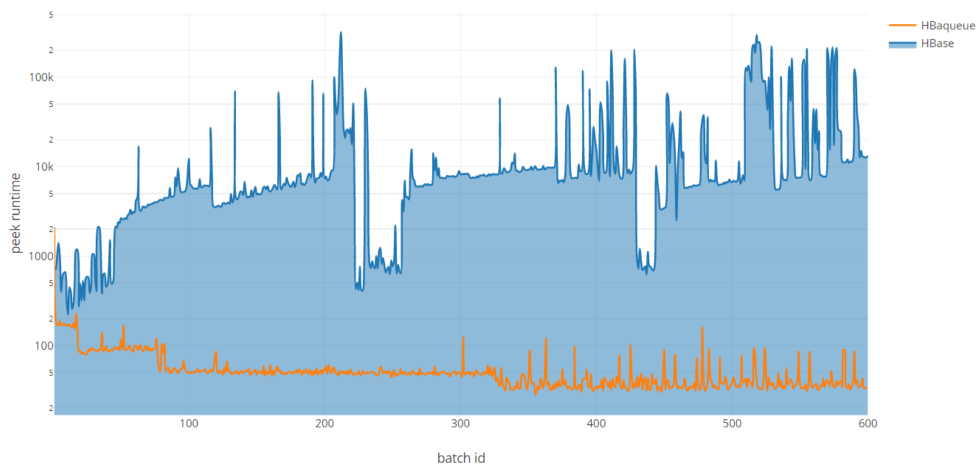


Figure 5.9: Batch-Dequeueing Latency in HBaqueue and HBase

5.4 Summary

In this chapter, we studied how current elastic streaming processing engines react to changes in the workload and showed the functionality of relevant components in a general architecture. To realize elasticity in distributed streaming pipeline, we proposed our elastic queue middleware (EQM) concept as a scalable locality-affinity buffer by wrapping HBase and exposing four elasticity methods to elastic streaming engine integration, thus significantly reducing the software engineering costs.

HBaqueue as an implementation of EQM based on HBase was further presented which tackles critical performance problems. Experimental results proved that the read performance in HBaqueue outperforms HBase

independently of growing storage size. The median write performance is 17% better and overall more stable as compared to the default HBase setting, thus allowing HBaqueue to give more strict performance guarantees to clients. It scales faster and better than HBase thus utilizing the cluster resources (namely disk space, network and computing power) more efficiently.

6 Workload-aware Data Extraction

Previous Chapter 3, Chapter 4 and Chapter 5 mainly focused on the *transformation* phase of Extract-Transform-Load (ETL) processes. In Section 3.3, our (distributed) incremental ETL pipeline was introduced based on a specific consistency model. To meet the *consistency* requirement while accessing warehouse tables, Change-Data-Capture (CDC) processes run continuously to extract source deltas for ETL jobs and to construct input batches of various sizes depending on the real freshness needs that are specified by the incoming analytical queries.

As noted in Section 3.3, the *exact* freshness demands are not always guaranteed since the resources (CPU, Memory, I/O bandwith) in those autonomous data sources are applied for their own primitive workload and their Service-Level-Agreements (SLAs) (e.g., transaction throughput), not for the SLAs (e.g., freshness, expiration ratio) of analytical queries for data warehousing. In this chapter, we present our "Workload-aware CDC" solution which enables our "On-demand ETL" to allow the business analysts to *negotiate* with the original data source users for the SLAs of overall workloads. We illustrate the motivation of workload-aware CDC in Section 6.1 and analyse the problem in Section 6.2. The design and implementation details are introduced in Section 6.3 and experimental results are shown in Section 6.4.

6.1 Motivation

CDC techniques have been widely used in many data integration scenarios (e.g., data replication, data warehousing) where specific forms of data copies need to be maintained incrementally. For example, data replication is used in web companies to provide their customers with online services to guarantee 24 x 7 availability. Data replicas are incrementally maintained by the same updates that are carried out over primary data sets. These replicas are called *hot standbys*, which provide the systems with fault tolerance, and can be optionally used to answer read-only queries with sa-

tisfied data freshness. The CDC processes synchronize the updates from primary, committed Online Transaction Processing (OLTP) transactions (e.g., through transaction log files) to remote replicas.

Another classic scenario is online business data warehouses. Data warehouse tables are continuously refreshed by complex ETL jobs with the data changes that are captured by the CDC processes from heterogeneous data sources. With different refresh policies, specific ETL jobs could be constructed and executed, for example, immediately, if there is any update occurring on the source side (*eagerly*), or at any time there is an OLAP query issued (*deferred/on demand*), or (*periodically*) in regular cycles.

In both scenarios, the CDC processes require a certain level of initial integration cost on the source side while, at runtime, they compete for resources with the original source-local (transaction) workload, which incurs negative impacts. Depending on the implementation of the CDC processes (e.g., log-based, trigger-based and timestamp-based) and process configurations (e.g., number of concurrent CDC threads and execution frequencies), the performance impact could be different.

With the flexibility of specifying freshness demands or execution time windows in Online Analytical Processing (OLAP) queries, modern application scenarios (e.g., Internet of Things (IoT), real-time dashboard) raise more critical requirements of acquiring business values in real time due to high competition in markets, thus forcing the analytical results to be returned in a more accurate, rapid manner. Notably, this pressure has been propagated back to the origin, which forces the CDC processes to run more efficiently to deliver accurate, just-in-time changes for ETL jobs. More formally, data changes of certain accuracy (freshness) have to be present in the staging area for ETL jobs to refresh relevant warehouse tables within strict deadlines. However, as a side effect, heavy CDC tasks would affect local OLTP workloads as resources are limited, which influences the *quality of service* (QoS) for OLTP users (e.g., transaction throughput). Therefore, the OLTP workload and the CDC workload should be scheduled together in terms of resource utilization to meet the QoS[12] for both applications.

More importantly, varying freshness and deadlines can be defined in online OLAP queries depending on different business analytics needs while the configuration settings of the CDC processes are normally deployed at system-initialization time without modification or adjustment at runtime. For example, if a CDC process was initially configured for

[12]We denote QoS for OLTP workloads as *Quality of Transactions* (QoT) and QoS for OLAP queries as *Quality of Analytics* (QoA).

high efficiency, most of the CDC requests are returned with very low latencies (at millisecond level). This would result in low transaction throughput even if the current OLAP queries are set with loose deadlines (e.g., seconds). In this case, the CDC processes run unnecessarily fast, ignoring that warehouse users may tolerate stale query results or high query response time. As another example, a *lightweight* CDC setting in favor of original OLTP users might have much less impact on local OLTP workloads whereas it is definitely not able to meet more demanding QoA metrics set by the business analysts. Given spikes in the CDC requests with high freshness demands or very *terse* deadlines, it is much likely that most of the requests expire or are returned with stale results. In a nutshell, the CDC requests can either be over-provisioned with system resources (which is a resource wastage for the CDC requests with late deadlines/low freshness needs) or starved (high expiration ratio for the CDC requests with early deadlines/high freshness demands). The crux is the unawareness of the diversity of QoA metrics at runtime with immutable, initial CDC setup.

Therefore, we propose a *workload-aware* CDC approach, which is more attractive for QLAP queries/CDC requests with varying deadlines and accuracy properties. The objective of this work is to examine the possibility and effectiveness of extending three existing, classic CDC methods (i.e., trigger-based, log-based, and timestamp-based) to workload-aware variants. A key challenge is to make these CDC methods be capable of adapting themselves to varying freshness demands with given deadlines at runtime. If recent source deltas in the staging area always deliver higher accuracy than the actual needs of incoming OLAP queries, the CDC processes would be scheduled less frequently, thus releasing resources for local OLTP workloads. If the data in the staging area is not always fresh enough, the scheduler would request the CDC processes to apply for more resources to capture the missing deltas in short time window, which affects the concurrent transaction executions.

To reach this objective, the execution configurations of the CDC processes must be adaptable to various QoA metrics at runtime, i.e., increase/decrease the scheduling frequencies or the execution parallelism. We denote the CDC configurations/settings as *gears* (as the behaviour is similar to *gear switching* for running automobiles).

6.2 Problem Analysis

In this section, we first introduce the semantics of the symbols used in this chapter. Based on the terminology, we formalize the problem and present a metric (called *performance gain*) to quantify the benefits of balancing QoT and QoA for the entire workloads.

6.2.1 Terminology

The entire end-to-end latency of delta synchronization starts from capturing source deltas, buffering them in the staging area (called CDC delay) and ends by merging the final calculated deltas with target warehouse tables. In this work, we only focus on the CDC delay without considering the transformation and loading phases as they would not influence QoT. Specific symbols are listed in Table 6.1.

Table 6.1: Symbol Semantics

Symbol	Description
$t_c(T_i)$	Committed time of transaction T_i at source
$t_r(T_i)$	Recorded time of transaction T_i in staging area
$t_a(C_i)$	Arrival time of CDC request C_i
$t_w(C_i)$	Time window of CDC request C_i
$d(C_i)$	Deadline of request C_i. i.e. $d(C_i) = t_a(C_i) + t_w(C_i)$
$t_{f_s}(t)$	Data freshness of OLTP source at time t
$t_{f_d}(t)$	Data freshness of staging area at time t
$t_{f_s}(C_i)$	Freshness demand of CDC request C_i
$S(C_i)$	Staleness of CDC request C_i
$L(C_i)$	Latency of CDC request C_i
S	Average staleness of CDC requests
L	Average latency of CDC requests
Er	Expiration ratio of CDC requests
τ	Transaction throughput of OLTP source
G	Overall gain of the systems (both OLTP and OLAP)
α	Normalization coordinate for OLTP workloads
β	Normalization coordinate for CDC workloads
w	QoT weight
$1 - w$	QoA weight

A source-side transaction T_i successfully commits at time $t_c(T_i)$ in an OLTP source system while it is extracted and recorded at time $t_r(T_i)$ in

the staging area. In this example, the freshness of the OLTP source at time t is denoted as $t_{f_s}(t)$ which is the latest commit time of the transactions in the OLTP system, i.e. max($t_c(T_i)$). The freshness of data staging area at time t is denoted as $t_{f_d}(t)$ which is the newest commit time of the updates captured in the staging area, i.e. max($t_r(T_i)$).

Furthermore, a CDC request C_i arrives at time $t_a(C_i)$ with a time window $t_w(C_i)$. This time window is the maximum time period which the C_i can tolerate to wait for the CDC to deliver the complete set of deltas, thus avoiding infinite suspension of OLAP queries due to limited resources in OLTP systems. The deadline $d(C_i)$ for the request C_i is calculated as the sum of the request arrival time $t_a(C_i)$ and the time window of the request $t_w(C_i)$.

In addition to the deadline specified in the request C_i, business analysts can explicitly inform the CDC of the wanted accuracy $t_{f_s}(C_i)$ of the analytical queries by, for example, invoking ensureAccuracy(...) through JDBC connection [TPL08]. To fulfil the expected freshness, the CDC processes should guarantee the following conditions, i.e., when the maximal waiting period is up, the freshness of the staging area should be equal to or greater than the specified freshness demand in C_i. Otherwise, the CDC request expires and returns results with specific staleness.

$$t_{f_d}(t) \geq t_{f_s}(C_i) \text{ and } t \leq d(C_i)$$

The staleness of request C_i is $S(C_i)$, the difference between the expected freshness and the actual freshness returned by C_i. If C_i is successfully fulfilled, the difference is zero.

$$\begin{aligned} &S(C_i) = t_{f_s}(C_i) - t_{f_d}(t) \text{ and } t = d(C_i) && (C_i \text{ expires}) \\ &S(C_i) = 0 && (C_i \text{ succeeds}) \end{aligned}$$

The latency $L(C_i)$ is the time taken by the CDC to answer the request C_i. If the request expires, $L(C_i)$ is equal to the time window of C_i.

$$\begin{aligned} &L(C_i) = t_w(C_i) && (C_i \text{ expires}) \\ &L(C_i) = t_r(T_i) - t_a(C_i) && (C_i \text{ succeeds}) \end{aligned}$$

To illustrate this concept with an example, Table 6.2 shows three transactions T_1, T_2, T_3 committed in an OLTP source system at points in time 10, 12 and 15, respectively (we consider natural number as the unit of time, zero as the oldest time). Their recorded time in the staging area is 11, 15 and 22, respectively (as shown in Table 6.3).

Table 6.4 illustrates three CDC request C_1, C_2, and C_3 which arrived at time 10, 13 and 18 with time window 1, 3 and 3, respectively. C_1 arrived at time 10 while the latest transaction committed at OLTP source is T_1 (committed at time 10). Therefore, its freshness demanded $t_{fs}(C_1)$ is 10. Similarly, $t_{fs}(C_2)$ and $t_{fs}(C_3)$ are 12 and 15, respectively. Deadlines of these CDC requests, i.e., $d(C_1)$, $d(C_2)$, and $d(C_3)$ are 11, 16 and 21, respectively.

Table 6.2: Transactions in OLTP Source System

Transactions	T_1	T_2	T_3
Commit time $(t_c(T_i))$	10	12	15
Freshness$(t_{f_s}(t))$	10	12	15

Table 6.3: Transactions arrived in Data Staging Area

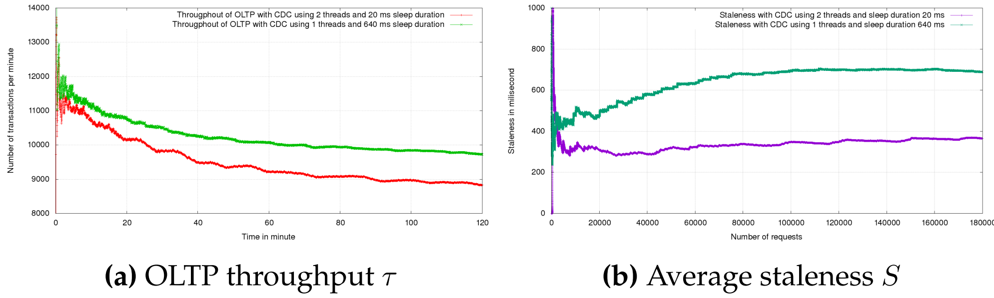
Transactions	T_1	T_2	T_3
Record Time $(t_r(T_i))$	11	15	22
Freshness $(t_{f_d}(t))$	10	12	15

Table 6.4: CDC Requests arrived in CDC

CDC requests	C_1	C_2	C_3
Arrival time $(t_a(C_i))$	10	13	18
Time window $(t_w(C_i))$	1	3	3
Deadline $(d(C_i))$	11	16	21
Freshness demanded $(t_{f_s}(C_i))$	10	12	15
Freshness returned $(t_{f_d}(t))$	10	12	12
Expired?	No	No	Yes
Staleness $(S(C_i))$	0	0	3
Latency $(L(C_i))$	1	2	3

When C_1 arrived at CDC at 10, the freshness demanded is 10 while the freshness in staging area $t_d(T_i)$ is less than 10. After one time unit, which falls within the time window $t_w(C_1)$ = 1, the freshness in the staging area reaches 10, thus C_1 can be returned successfully. Therefore, the staleness $S(C_1)$ is zero and the latency $L(C_1)$ is 1.

For the CDC request C_2 (occurring at 13), it took 2 time units (at 15) for the complete deltas of the transaction T_2 to become available in the staging area, which is still less than the deadline of C_2, i.e., 16. Hence, the latency is 2 and the staleness is 0.



(a) OLTP throughput τ **(b)** Average staleness S

Figure 6.1: QoT vs. QoA using Trigger-based CDC

The CDC request C_3 has its time window as 3 and the deadline as 21. However, the expected freshness (which is 15) appeared in the staging area only at 22, which exceeds the deadline. C_3 expired and the actual freshness returned is 12. Here the staleness is the difference between the expected freshness and the received freshness, which is 3, and the latency is the entire time window, which is also 3.

The effectiveness of the OLAP systems and the CDC processes is represented by the average QoA (Quality of ETL) of recent CDC requests. It can be defined by three measures: average staleness (S), average latency (L), and expiration ratio (Er) (see Table 6.1). For example, in the above example, the expiration ratio Er is 1/3 = 33.3%.

As compared to QoA metrics, we use notion τ to represent the average QoT of the OLTP source systems. Since the OLTP applications are characterized by a large number of short online transactions (with INSERT, UPDATE, DELETE statements), the notion τ denotes the transaction throughput which is the number of transactions processed in a minute in an OLTP source system.

To better illustrate the impact of CDC processes on original OLTP systems, we conducted an experiment which compared the QoT metric τ with the QoA metric S (average staleness) using trigger-based CDC with two configuration settings: high *gear* - 2 threads + 20ms scheduling period vs. low *gear* - 1 thread + 640ms period. As shown in Figure 6.1, although the average staleness S in the high gear performs better than that in the lower gear, the transaction throughput τ is way worse and not satisfactory for OLTP workloads, i.e. delivering fresher deltas incurs significant overhead on transaction executions. This experiment demonstrates the trade-off between QoT and QoA and the necessity of a workload-aware CDC for trade-off balancing.

6.2.2 Performance Gain

In order to quantify the effectiveness of our workload-aware CDC, we analyse the QoT and the QoA together and introduce a metric called *performance gain G*, as shown in the following equation. The trade-off between QoT and QoA is represented in the equation as well.

$$Gain(G) = w \times \alpha \times \tau + (1 - w) \times \beta \times \frac{1}{S \times L \times Er}$$

Since τ drops with shrinking staleness S, latency L or expiration ratio Er, the QoT value is directly set as τ while the QoA value is calculated as the reciprocal of the production of S, L and Er. In order to compare QoT and QoA values together, we introduce two normalization coordinates α and β for QoT and QoA, respectively (see Table 6.1). In addition, users have the flexibility to set weights for both qualities of services, i.e., w, which can be adjusted at runtime. For example, users can prioritize fresh reports/dashboards at the beginning of the day and high OLTP throughput during working hours.

The performance gain value G is formed as the sum of weighted, normalized values of both QoT and QoA. The objective of workload-aware CDC is to approach max(G) at any time.

6.3 Workload-aware CDC

In this section, we present our workload-aware CDC approach. We first introduce the global architecture with several key components. Furthermore, we describe the workload-aware extensions of three CDC implementations: trigger-based, log-based and timestamp-based methods with their *gear* settings.

Figure 6.2 shows the architecture of workload-aware CDC. At the left side, there are three data sources extended for specific CDC approaches: trigger-based, timestamp-based and log-based, respectively. Source deltas are extracted by the workload-aware CDC components (in the middle of this figure) in the ETL processes and propagated from each source to an ETL staging area or a message broker, e.g., Kafka (at the right side).

Each CDC component runs inside an ETL process, which is composed of an *agent* thread, a task queue of regulable size and a *worker* thread pool. Depending on the source-side CDC extension, the agent thread generates CDC tasks for specific delta ranges at a rate which can be customized at runtime (Step 1). Every CDC task is put immediately in the task queue

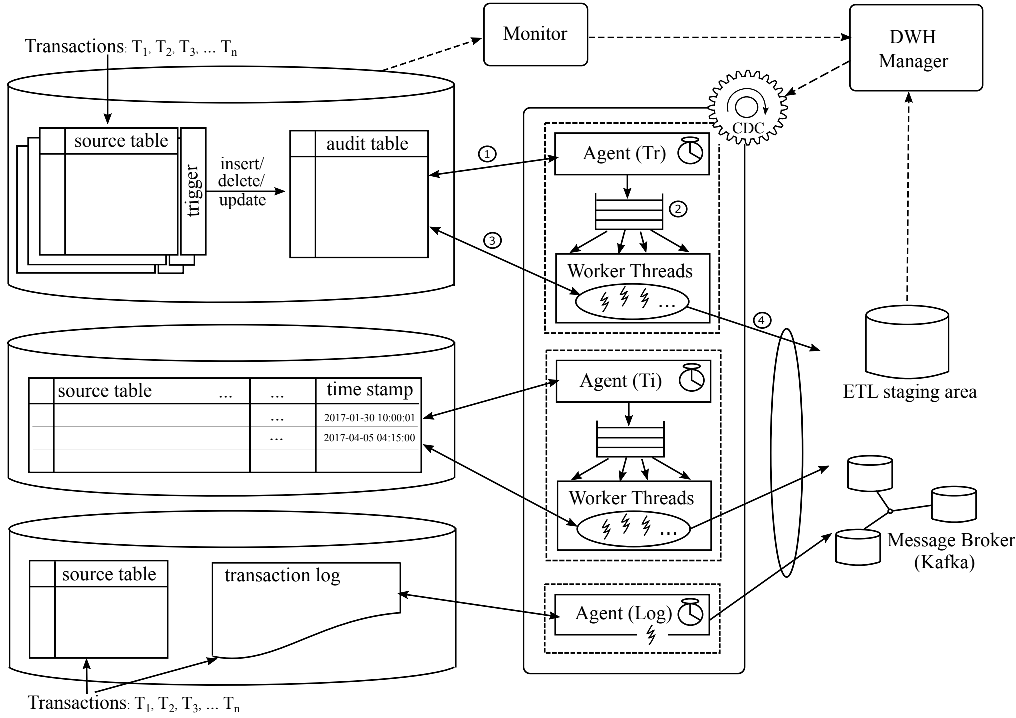


Figure 6.2: System Architecture for Workload-aware CDC

(Step 2) and further executed to fetch (Step 3) and output (Step 4) source deltas, once the worker thread pool has an idle thread. From this setting, we see that, the more frequently the agent generates CDC tasks, the busier are the worker threads with reading source deltas and the higher is the freshness in the ETL staging area.

To achieve workload awareness, the traditional Data Warehouse (DWH) manager is extended with additional workload-aware CDC logic. The main body of the logic lets the DWH manager to keep calculating the performance gains (see Subsection 6.2.2) based on current QoT and QoA. The QoT, i.e., transaction throughput τ, is continuously reported to the DWH manager by a source-side *monitor* while the QoA metrics, i.e., average staleness S, latency L and expiration ratio Er, are measured over the final status of the CDC requests sent to the DWH manager. With varying freshness demands and deadlines set in incoming CDC requests, the DWH manager triggers (e.g., raising/shrinking frequency or resource pool size) the CDC executions to switch to new gears immediately, once the performance gain drops.

In the following, we look into details how specific CDC methods are extended towards workload-aware variants.

Workload-aware trigger-based CDC. The extension for trigger-based CDC in a source database is shown at the upper-left corner of Figure 6.2. Triggers are functions which are defined manually to specify how the updates are tracked from committed transactions over source tables and fired automatically whenever certain events occur (e.g., after/before Insert, Delete, Update statements). Captured updates are recorded in a separate table called *audit* table and become visible only when the original transactions commit.

The corresponding workload-aware trigger-based CDC component (at the upper-middle of Figure 6.2) of an ETL process fetches changes/deltas directly from the source-side audit table by issuing SQL queries. In order for the agent thread to identify new deltas, the audit table schema is designed to track the sequence ids of transactions. The agent thread periodically reads the newest sequence id in the audit table and compares it with the cached maximum sequence id read from the last time. If a new sequence id is found, a task is constructed with a sequence id range (old id, new id], i.e., lower and upper bounds of the sequence ids of those new delta tuples that need to be fetched from the audit table. When being processed by a worker thread, a delta-fetching SQL query is generated with this sequence id range as a predicate condition in the WHERE clause.

Gears of the workload-aware trigger-based CDC are characterized by two properties: *scheduling period* (i.e., how long the agent stays idle in a period) and *worker parallelism* (i.e., the size of the worker thread pool). High gear is set with short scheduling period and high worker parallelism, which keeps the deltas up-to-date in the ETL staging area. However, it imposes more SQL-query load over the audit table and contends resources and locks with original OLTP transactions. Low gear is set with long period and low worker parallelism, which influences the sources in the opposite way.

Workload-aware timestamp-based CDC. Under trigger-based CDC in Figure 6.2, the source-side timestamp-based CDC extension is shown. Instead of using an audit table, the original source table is extended with an extra *timestamp* column. Its value is updated whenever the corresponding tuple is inserted or updated. Modifications of tuples are tracked directly over the source tables based on newer timestamps. However, to quickly identify deltas, the timestamp column is indexed, thus introducing extra index-maintenance overhead for OLTP transactions as similar to audit tables. Besides, deletions cannot be tracked since the timestamp information will be deleted together with the tuples themselves. Furthermore, intermediate updates of a tuple could be missed during two CDC runs as the timestamp value could be overwritten by new ones at any time before

the change is captured.

Similar to the trigger-based counterpart, the agent of the workload-aware timestamp-based CDC component caches the latest timestamp recorded from the last run and compares it with the maximum timestamp found in the current run. The sequence id range is replaced with a timestamp range in the WHERE clause of a delta-fetching SQL query.

The gear settings are the same as those used in the trigger-based CDC and so are the influences over original sources. In addition, the more frequently the CDC runs, the more deltas are captured from a tuple as old versions would not be missed. This also leads to more concise deltas but more network bandwidths occupied at the source side.

Workload-aware log-based CDC. At the bottom-left corner of Figure 6.2, we see the log-based CDC approach. General databases use transaction logs (usually write-ahead logs (WAL)) for recovery purpose in the cases of database crash and power outage. Before running an INSERT/UPDATE/DELETE statement of a transaction, a log record is created and written into an in-memory WAL buffer. As the transaction commits, all buffered log records are encoded as binary stream and flushed (through *fsync*) into a disk-based WAL segment file (a WAL contains multiple segment files). Log records of concurrent transactions are sequentially written into the transaction log in commit order and additional optimization work batches up log records from small transactions in one *fsync* call, which amortizes disk I/O.

The log-based method reads the WAL segment files sequentially from disk (with I/O costs) and decodes the binary WAL stream through vendor-specific API into a human-readable format (using extra CPU cycles). Sequential read is fast as it saves disk seek time and rotational delay. With a *read-ahead* feature supported by most of the operating systems, subsequent blocks are prefetched into the page cache, thus reducing I/O.

As compared to the trigger-based and timestamp-based CDCs, the main benefit of using log-based CDC is that there is no specific CDC extension needed (i.e., no audit table or timestamp column) at the source side, which eliminates the resource/lock contentions. The limitation of log-based approach is that the log entries can only be read out in sequence, thus prohibiting CDC worker parallelism. Therefore, gears of workload-aware log-based CDC are only determined by the scheduling period set in the agent thread, which, for example, affects source-side I/O bandwidths.

6.4 Experiments

In this section, we evaluate the performance gain of our workload-aware CDC approach against traditional CDC settings, which are in turn reflected by QoT and QoA metrics, i.e., average latency L, staleness S, expiration ratio E for CDC requests and throughput τ for transactions.

6.4.1 Experiment Setup

The test bed is a 6-node cluster which is composed of machine nodes (2 Quad-Core Intel Xeon Processor E5335, 4x2.00GHz, 8GB RAM, 1TB SATA-II disk, Gigabit Ethernet) in different roles. One node played the role of source database which hosted a PostgreSQL [pos] database. An open-source TPC-C [tpca] benchmark called BenchmarkSQL [ben] was deployed in one of the rest nodes and ran continuously to simulate and impose OLTP workload over the source database. Three of the cluster nodes were selected to simulate the CDC processes with the same settings and competed for resources with the BenchmarkSQL processes on PostgreSQL node. The last node in the cluster ran a CDC-request simulator process which kept issuing CDC requests with varying freshness demands and deadlines, which distinguished different priority patterns in different time slots.

In each test run, we selected a different CDC setting and observed the above-mentioned four metrics for one hour. The results of three CDC approaches are analyzed in the following.

6.4.2 Trigger-based CDC Experiment

For trigger-based CDC, we compared the results in four CDC settings: three constant settings, i.e., *high gear* (scheduling period 20ms, thread pool of size 2), *middle gear* (160ms period and 2 threads), *low gear* (1 thread with 320ms period) and a workload-aware scheme with mixed gears.

As shown in Figure 6.3, we see that neither the high gear nor the low gear was able to balance the QoS for both OLTP and OLAP workload. Using high gear, almost all the CDC requests were finished in time with the shortest (20ms) latency and kept the deltas most up-to-date in the staging area (only ~10ms delay) while the transaction throughput τ is the lowest (around 36K/min). The low gear shows the opposite extreme case where the τ reached the highest value (40K/min in average), but the QoA metrics were not satisfactory at all (almot 50% CDC requests got expired with very high staleness).

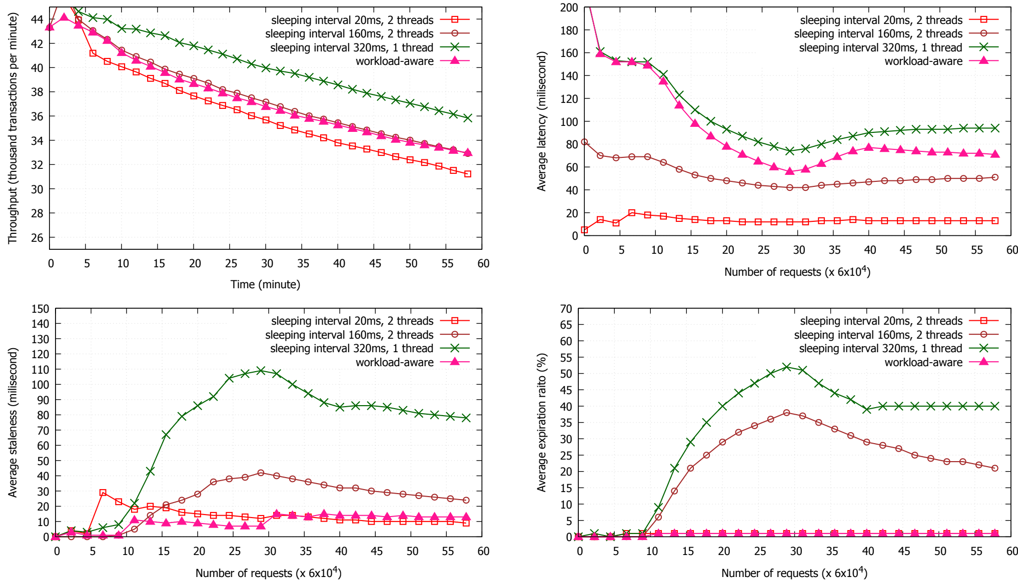


Figure 6.3: QoT & QoA of Trigger-based CDC Variants

The middle gear served as a compromise between the high and low gears where the τ (39K/min), the latency L (60ms) remained the median values between two extremes. However, the staleness S (30ms delay) and the expiration ratio E (30%) were still not acceptable.

Our workload-aware CDC outperformed the middle gear in terms of the S and the E, i.e., they are the same as in the high gear. The τ was the same as in the middle gear, even though the L was slightly higher (80ms). This fact shows exactly that the workload-aware scheme took the source-side impact into consideration and exploited the deadlines of incoming CDC requests to extent while still delivering high delta freshness in time. The reason behind this fact is that it was able to stop the downtrend of the performance gain by switching to appropriate gears at runtime.

6.4.3 Log-based CDC Experiment

As described in Section 6.3, log-based CDC cannot exploit worker parallelism due to sequential read of log files, hence we only compared CDC settings among two constant scheduling frequencies: *high gear* (0ms) and *low gear* (120ms), respectively, and a workload-aware scheme which changes the frequency between these two gears on demand. Figure 6.4 depicts the experimental results of log-based CDC.

Similar to the experimental results of trigger-based CDC, the log-based

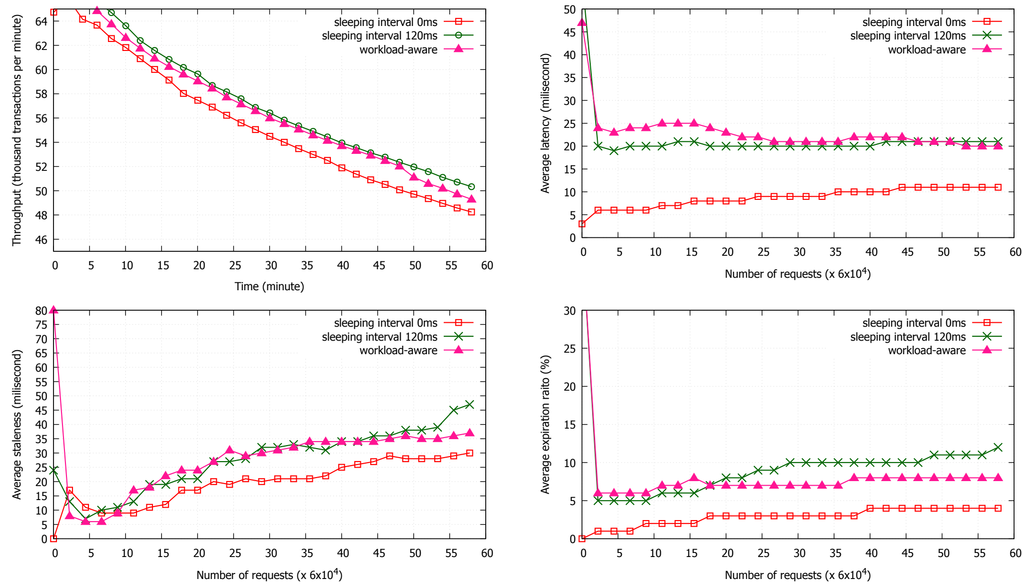


Figure 6.4: QoT & QoA of Log-based CDC Variants

CDC processes in the high gear and the low gear showed two extreme cases: the τ was better in low gear while all L, S and E were better in high gear. In contrast, our workload-aware log-based CDC was able to achieve almost the same τ as in the low gear while keeping most of the incoming CDC requests achieving the median staleness and expiration ratio. We see that the overall QoS can still benefit from the workload-aware CDC approach even without considering worker parallelism.

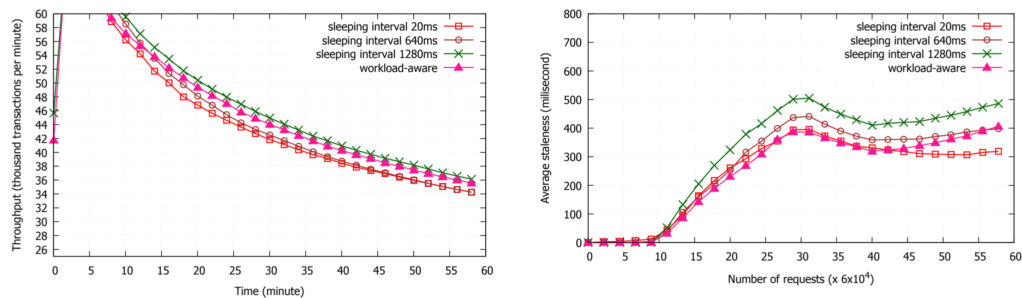


Figure 6.5: QoT & QoA of Timestamp-based CDC Variants

6.4.4 Timestamp-based CDC Experiment

Figure 6.5 shows the experimental results of four timestamp-based CDC settings (*high gear*: scheduling period 20ms, *middle gear*: 640ms, *low gear*: 1280ms, a workload-aware setting with mixed gears).

The τ in the workload-aware setting is very close to the highest throughput which matches the case of the low gear while the average staleness E remained most of the time, as low as the lowest one calculated from the CDC executions in the high gear. These results match the same conclusions derived from the trigger-based and log-based CDC variants shown above. With dynamic CDC settings customized at runtime, the performance gain in the workload-aware mode led all the way against the other three constant CDC settings throughout the entire experiment.

6.5 Summary

In this chapter, we addressed the challenge of making the change data capture (CDC) processes be aware of varying freshness demands and deadlines specified in the OLAP queries while meeting the SLA requirements on the original data source platforms. We proposed a self-adaptive CDC approach which is able to adjust the so-called CDC gears at runtime based on the performance gain as the feedback tracked from both the OLTP and OLAP workload. From the experimental results we see that our workload-aware CDC approach is able to react to varying QoS results at runtime and balance the overall QoS for both OLTP and OLAP workload to extent.

7 Demand-driven Bulk Loading

In this chapter, we address on-demand data loading in large-scale data analytics by looking into the "hotness" and "coldness" of analytics data sets. Based on observations derived from data access patterns and query distributions in real social graphs, we exploit a trade-off between fast data availability and low query latency and propose our "Demand-driven Bulk Loading" (DeBL) scheme.

We describe the motivation of DeBL in Section 7.1. The architecture and implementation details are present in Section 7.2 and the experimental results are analyzed in Section 7.3.

7.1 Motivation

As the biggest social network company, Facebook's social graph system nowadays serves tens of billions of nodes and trillions of links at scale [CBB+13]. Billions of daily queries demand low-latency response times. A social graph benchmark called LinkBench [APBC13] was presented by Facebook which traces real distributions on both data and queries on Facebook's social graph stores.

We learned several interesting facts from LinkBench, for example, one observation on access patterns and distributions states that there is always some "hot" data that is frequently accessed while massive amounts of "cold" data is seldom used. With a 6-day trace, 91.3% of the data is cold while hot data often exists around social graph nodes with high out-degrees. For example, a video with high *like* rates will be recommended more widely than others. Moreover, 50.7% of the queries contain a so-called *get_link_list* request which searches for "what objects have been frequently liked by a user". Such type of requests performs a short range scan in massive amounts of rows in graph stores based on user ids.

To provide deep insights on user activities, processes of generating daily analytical reports or training business/user models are scheduled offline after massive source data sets (e.g., user click streams) are loaded into analytical (graph) warehouses within an off-peak time window (e.g., at night).

When bulk-loading daily social graphs to analytics platforms, the availability of hot data is delayed since the analytics will only be scheduled when all data has been loaded. For frequently emerging queries, analytics platform uptime could start earlier if there was a mechanism that can tell whether all the relevant data is already available before the remaining data is loaded. As data is loaded chunk by chunk, it would be enough to have a global view of the key ranges of all the chunks before starting loading. This can be seen as an index construction process. In this way, a query can identify its relevant chunks by comparing its queried key with the key ranges of all the chunks. Once its relevant chunks have been loaded, this query can start to run over the current storage state. In addition, as the small amount of hot data can be arbitrarily distributed among all the chunks, faster availability to frequent queries can be achieved by prioritizing the loading of the chunks which contain hot data.

As detailed in Subsection 2.7.4, for both traditional databases (e.g., MySQL [mys]) or classic NoSQL databases (e.g., HBase [Geo11]), bulk-loading costs are similar since all files need to be sorted (either to build a B-tree index or to generate HFiles). But HDFS's batch processing features can be exploited in HBase's two-phase bulk loading approach to build indices on copied chunks in batch upfront before going to the second transformation phase, which is more desirable.

However, according to the experimental results in [CST+10, RGVS+12, APBC13], the performance of the most frequent *get_link_list* request in MySQL slightly outperforms that in HBase (~2x faster). Therefore, we see a trade-off between fast availability and low query latency here. Loading all data in HBase can have fast availability by creating indices upfront. Loading all data in MySQL leads to low query latency after long-time bulk loading ends. It makes sense to load only a small amount of hot data into MySQL in a smaller time window for fast processing while copying massive amounts of cold data into HBase where its query latency is still acceptable for cold data. But the cost of identifying the hotness and coldness of tuples could be a large overhead for bulk loading. Hence, we introduce our "Demand-driven Bulk Loading" scheme to address these considerations.

7.2 Architecture

In this section, we introduce the architecture of our demand-driven bulk loading scheme. According to the modification timestamps of files, the whole input data set is separated into two parts: small number of recent

files and large, historical files. Recently changed files (*hot data*) are imported into a MySQL table called "link table" using MySQL's own fast bulk load utility. These files are used for answering queries on hot data and providing partial results for all other queries. Meanwhile, massive amounts of historical files (*cold data*) are first split to chunks with predefined size and then copied into HDFS in batch. With parallel loading of recent and historical files into MySQL and HDFS, respectively, the latency of loading HDFS is normally higher than that of MySQL's bulk load due to the input size, thus dominating the overall load speed. After loading in MySQL completes, hot data is available for querying. The HDFS files will be gradually loaded into HBase to complement the MySQL query results for answering queries that involve cold/historical data. Figure 7.1 illustrates the architecture of our hybrid-storage approach.

Two main processes are involved in this hybrid-storage architecture: offline index building and online bulk load coordination. In contrast to traditional bulk load approaches, client requests are allowed to query link data before the historical data sets are completely available in HBase. To determine the completeness of query results on the client side, a so-called *bucket index table* is used. At the HBase layer (left) side of this architecture, an offline MapReduce job called *distribute chunk* (*dist_ch*) job is batch processed on each file chunk in HDFS once copied from the remote server by a HDFS loader. The implementation of this job is based on a hash function that maps and writes each text line in a chunk to a specific "bucket" (HDFS file) with a unique id and a key range disjoint from others. A new bucket will be created if needed and its *bucket index* and *key range* will be captured by the bucket index table at the (right-side) MySQL layer. These steps form the *offline index building* process. More details will be provided in Subsection 7.2.1. With the completion of the last *dist_ch* job, all cold files have been copied into HDFS and clustered into multiple buckets with disjoint key ranges. The key ranges and index information of all the buckets are contained in the bucket index table in MySQL.

At this time, system uptime begins and query requests are allowed to run on our hybrid link data storages through a client-side *coordinator* component. At the same time, another online MapReduce job called *generate HFiles* (*gen_HF*) job is triggered to continuously transform buckets in HDFS to HFiles so that they can be randomly read from HBase. The transformed HFiles incrementally build a "link table" in HBase which can be seen as an external "table partition" of the "link table" in MySQL. Tuples stored in both engines share the same logical table schema. With a given key (the user id) specified in the *get_link_list* query, the coordinator checks whether the HBase layer should take part in this query execution

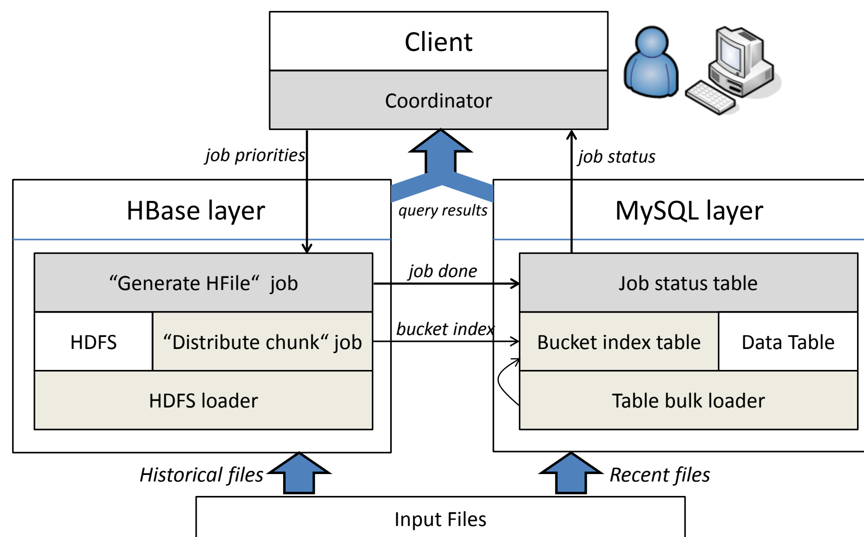


Figure 7.1: Architecture of Demand-driven Bulk Loading

by asking the MySQL-side bucket index table. If so, another *job status table* tracks the availability of the required tuples in HBase. For tuples available in HBase, the query request is offloaded to HBase by the coordinator. In case there are tuples that are not available yet because they reside in buckets that wait in the *gen_HF* queue, the query is marked as "incomplete" and buffered by the coordinator. As more and more incomplete queries occur at the coordinator side, the coordinator makes the online MapReduce job prioritize the job execution sequence for specific buckets, delivering fast availability on demand. Once the buckets are transformed to the portions of HBase's "link table", corresponding buffered queries are released. This process is called *online bulk load coordination*. The implementation of this process will be detailed in Subsection 7.2.2.

7.2.1 Offline Loading Index Construction

We use a dedicated Hadoop cluster to take over the job of loading massive amounts of cold/historical link data from remote servers to MySQL. HDFS's free copy/load speed and batch processing (using MapReduce) natures are exploited here to provide only indices on clustered file groups (buckets) using the *dist_ch* jobs, as introduced above. A *dist_ch* job writes text lines in each chunk in HDFS to different buckets (HDFS directories)

and outputs indices of new buckets to MySQL's bucket index table[13].

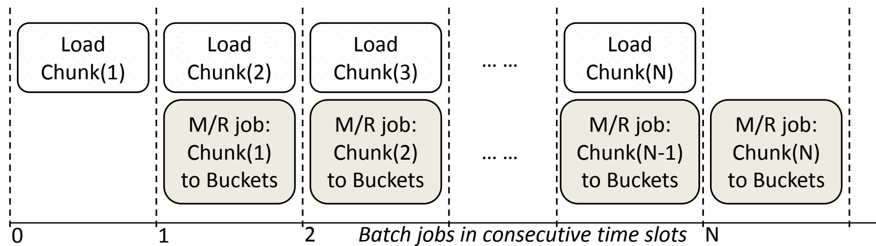


Figure 7.2: Execution Pipelines in the HDFS loader

Instead of running one big MapReduce-based *dist_ch* job after all files have been completely copied from a remote server, large historical files are split to several chunks and multiple small *dist_ch* jobs are executed in parallel which copy small file chunks to HDFS. As shown in Figure 7.2, chunk copying and *dist_ch* job run simultaneously in each time slot except the first and the last one which copies the first chunk and builds the indices for the last chunk, respectively. The resource contention is low since chunk copying does not use any MapReduce job and each *dist_ch* job runs individually. As the chunk copying pipeline overlaps the *dist_ch* job pipeline, the overall latency is derived from loading all chunks plus running the last *dist_ch* job. The chunk size is selected in a way that the latency of loading a chunk of this size is higher than running one *dist_ch* job on that chunk. If this requirement can be guaranteed, the chunk size should be defined as small as possible so that the time running the last *dist_ch* job is the shortest. Therefore, the overall system downtime is similar to the latency of copying massive amounts of cold data from a remote server to HDFS.

As tuples belonging to a specific key might be arbitrarily distributed in all chunks, we introduce a simple hash function to cluster tuples into buckets according to disjoint key ranges. If we have numeric keys (e.g. user *id1* for links) and a key range of 1K (0..1K; 1K..2K; ...), the bucket index for each tuple is derived from $b_index = round(id1/1K)$. With bucket index and key range, the coordinator can tell exactly which bucket should be available in HBase to complete the results for an incomplete query. As shown on the left side of Figure 7.3, *dist_ch* jobs take fix-sized chunks as inputs and generate a set of buckets of dynamic sizes based on *id1*. This

[13]Hadoop's MultipleOutputs is used here to include TextOutputFormat and DBOutputFormat for writing lines to buckets and writing indices to MySQL, respectively

can be explained by the LinkBench's observation on the distribution of nodes' outdegrees described in Section 7.1. Here, a bucket might contain a large number of links which belong to a node with very high outdegree.

7.2.2 Online Loading and Query Coordination

Once the remote files are copied to local HDFS, our system starts to accept queries from the client side and a *gen_HF* job runs continuously to finish the remaining bulk load work. Three components are involved here: the client-side coordinator, two tables in MySQL (bucket index table and job status table) and the online *gen_HF* job in HBase layer (see Figure 7.1).

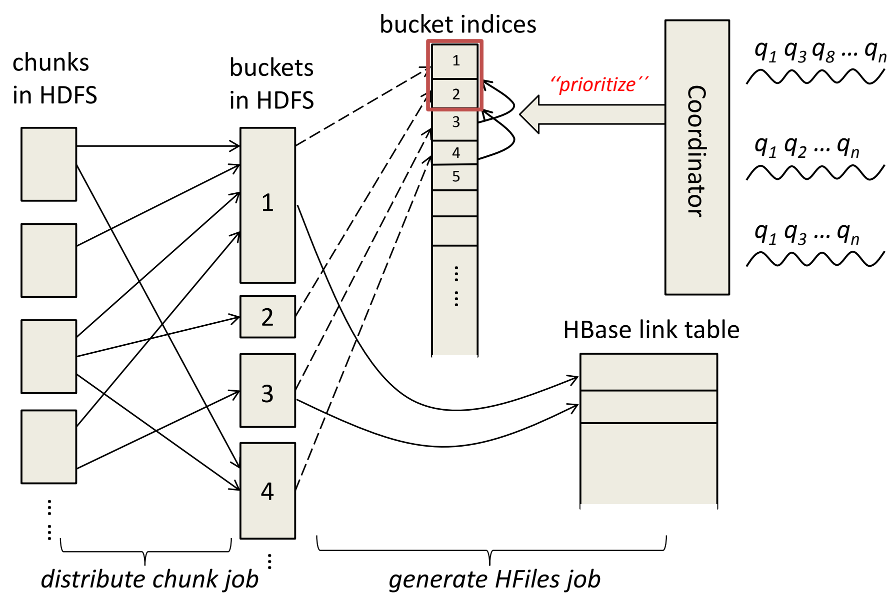


Figure 7.3: Offline & Online MapReduce Jobs

A *gen_HF* job is a MapReduce-based bulk loading job that is executed in several runs in the HBase layer. In each run, it takes HDFS files in "bucket" directories (directed by bucket indices) as input and generates HFiles as results (see Subsection 2.7.4). Tuples in HFiles can be randomly read without batch processing. The cost of the *gen_HF* job is dominated by sorting. When the local memory on each region server cannot hold the entire set of Put objects for in-memory sorting, expensive disk-based external sorting occurs. Hence, a *gen_HF* job each time will take only the top two buckets (included in the red rectangle in Figure 7.3) as input to avoid external sorting on local region servers.

The coordinator plays an important role in our demand-driven bulk loading scheme. It maintains a set of four-element tuples (*b_index*, *j_stat*, *k_range*, *q_list*) at runtime. The *b_index* is the bucket index which directs the *gen_HF* job to the input files in this bucket. The *k_range* represents the key range of these input files which will be further checked by an incoming query whether this bucket can contain required tuples. The *b_index* and *k_range* of all the buckets are initially read from the MySQL-side bucket index table at once. Note that, after the hot link data has been bulk loaded into MySQL, a special bucket will be created to contain the *k_range* of MySQL-side "link table".

Furthermore, before the *gen_HF* job starts a run, it registers its two input bucket indices in the job status table in MySQL. When the job is done, it updates its status in the job status table, whose content will be periodically pulled by the coordinator to maintain the *j_stat* elements for all buckets. At the beginning of system uptime, files in most of the buckets have not been transformed to HFiles and thus are not available in HBase. It's much likely that the incoming queries at that moment cannot be completely executed and are further pushed into the query list *q_list* of certain buckets. The coordinator will release the queries in a *q_list* once the *j_stat* states that this bucket is readable. Moreover, the coordinator will also sort the four-element tuples according to the size of *q_list* due to emergency so that the *gen_HF* job will always work on transforming the top two buckets with the largest number of waiting queries.

As an example of demand-driven bulk loading shown in Figure 7.3, three client-side threads keep sending queries to the coordinator. Most of them contain queries that would access tuples in the key ranges of bucket 1 and 3. The coordinator checks the sizes of the query waiting lists and prioritizes the *gen_HF* job execution sequence for bucket 1 and 3. Hence, after the next run, query q_1 and q_3 will be released by the coordinator since the required tuples now can be found in the HBase "link table".

7.3 Experiments

To enable fast availability (of "hot" data) in the entire social graph system, we introduced our "Demand-driven Bulk Loading" (DeBL) scheme. In this section, we validate our approach by analyzing the experimental results. The performance difference in terms of load speed and query latency is shown by comparing the results using a single MySQL database, using a single HBase cluster or using our DeBL approach. Our approach serves as a compromise between these two systems and outperforms both

of them when loading large-scale social graphs.

7.3.1 Experiment Setup

We used a logical *link* table with its schema (*id1*, *link_type*, *id2*, ..., *data*) to represent links stored in MySQL, HBase or both systems (as links occupy the largest portion in a graph, we ignored loading graph nodes in our test). The test query is the *get_link_list* operation which performs a short range scan to fetch links with given user *ids* and *link_types* and constitutes 50% of the whole workload. We think that this test setup is general and representative.

7.3.2 Loading Phase

We extended the LinkBench [APBC13] program for our test purpose which is based on a client/server architecture. In the original LinkBench implementation, multiple threads run on the client side to either load links (load phase) or send requests (query phase) to a server-side "link table" (using MySQL INNODB engine) in a MySQL database. To compare the load performance of different approaches, we first recorded the latency of bulk loading a CSV input file (100M links, 10GB) from a remote client (through 100Mb/s Ethernet) into a link table in a MySQL instance running on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM, 1TB SATA-II disk) using LOAD DATA INFILE command (the primary key index was first disabled and re-enabled after the file was loaded) [**bulkLoad (MySQL)**]. Since we were only comparing short range scan performance in the query phase later, a faster "link table" using MySQL MYISAM engine instead was used which is optimized for read-heavy operations (MySQL's batch insert was excluded in this test as MYISAM engine uses table-level locking on tables which is very slow for massive, concurrent inserts).

In the second case, we tested the bulk load performance on a HBase cluster - a 6-node HBase cluster (version 0.94.4) with a master running on the same node as the MySQL instance and 5 region servers (2 Quad-Core Intel Xeon Processor X3440, 4×2.53GHz, 4GB RAM, 1TB SATA-II disk) connected by Gigabit Ethernet. A big MapReduce job (the *gen_HF* job) was implemented to generate HFiles and populate a HBase link table after the same input file was copied from remote to local HDFS [**bulkLoad (HBase)**]. Since HBase also provides high write throughput, we also tested the performance of writing links to HBase in batch using HBase's Put

method [**batchPut (HBase)**]. To improve performance, the link table was first pre-split evenly in the cluster according to its key distribution to avoid "hotspots" during loading and querying.

Two variants of the load phases were tested using our DeBL approach. The first variant was composed of bulk loading 2GB, recently changed, remote link subsets into the MySQL table, copying the rest 8GB link files in batch from remote client to HDFS (in our HBase cluster) and meanwhile running multiple small MapReduce jobs (the *dist_ch* jobs) in parallel [**DeBL (2g mysql)**]. Another extreme case was shown by the second variant where no files were loaded into MySQL and the entire input file was copied to HDFS [**DeBL (no mysql)**]. In this case, the "hotness and coldness" in input data was not pre-defined manually but was captured automatically by our coordinator during *gen_HF* phase according to incoming query distribution. To indicate fast availability of DeBL approach, we attached the time taken to simply copy the test input file to server's local file system [**fileCopy**] to the final results as the bottom line as well.

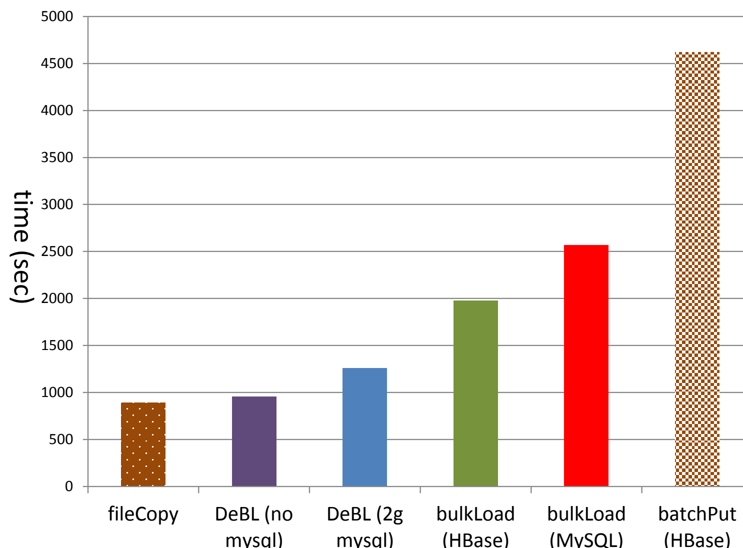


Figure 7.4: Elapsed Time during Loading Phase

The results of load latencies are shown in Figure 7.4 and detailed in Figure 7.5. The latency of **fileCopy** is the bottom line which is 893 seconds and cannot be improved anymore. The result of **DeBL (no mysql)** is 952.17s and very closed to **fileCopy**'s latency. The input file was transferred chunk by chunk and each chunk has 256MB size. With this chunk size, both *chunk-load* job and *dist_ch* job took similar time (24 25s). As both jobs ran in parallel, the result of **DeBL (no mysql)** could be derived from

the sum of **fileCopy**'s time and the latency of the last *dist_ch* job. It provides the fastest starting time of system uptime with near wire-speed. However, incoming queries still have to wait until their files are available in HBase. Another variant **DeBL (2g mysql)** took a little bit longer for bulk loading 2GB hot data into MySQL (including index construction) which is 463.26s and its total latency is 1257.23s.

DeBL (no mysql)		DeBL (2g mysql)		bulkLoad (MySQL)		bulkLoad (HBase)	
chunk load:	24.33	mysql load:	463.26	bulk load:	1674.76	HDFS copy:	895.31
dist_ch job:	24.95	hbase load:	793.98	gen. index:	890.57	gen. HFiles:	1082.81
total:	952.17	total:	1257.23	total:	2565.34	total:	1978.12

Figure 7.5: Detailed Latency in Loading Phase

Latency gets higher when using traditional bulk loading approaches. Using **bulkLoad (MySQL)**, the LOAD DATA INFILE command took 1674.76s while reenabling primary key index spent 890.57s. Using **bulkLoad (HBase)**, copying remote files to HDFS had the same latency as **fileCopy** and generating HFiles reached similar cost (1082.81s) as MySQL's index construction since both processes required sorting on large input files. However, **bulkLoad (HBase)** is faster than **bulkLoad (MySQL)** since HBase is a distributed system where MapReduce's batch processing feature can be exploited. Apart from this difference, both bulk loading approaches still outperforms HBase's fast writes where some overheads (e.g., compactions) occurred in HBase (see Subsection 2.7.2).

7.3.3 On-demand Data Availability

The end of load phase indicates the start of system uptime and lower load latency means faster data availability. After the load phase, we ran 50 threads on the client side to continuously send 500,000 *get_link_list* requests to the server. Both MySQL and HBases' bulk loading approaches led to complete data availability and we tracked the query throughputs starting from their system uptime. In this case, the difference of the query throughputs represents the difference of query latencies on MySQL and HBase as well. We tried our best to tune our HBase cluster, for example, by enabling block caching, setting up Bloom Filters and pre-splitting regions. For 10GB test data, our in-house HBase cluster yet could not cope with a single-node MySQL engine in terms of query throughput, as shown in Figure 7.6. However, as HBase cluster took less time to ingest the input data, its throughput curve started earlier (from 1978.12s, the end of its load

phase) to rise and converged at 278.7 op/sec throughput and 3.6 ms query latency in average for each *get_link_list* request. The **bulkLoad (MySQL) approach** took the longest time until all links are available in link table. Its throughput was rising rapidly (till 521.8 op/sec, 1.9 ms) and all the queries were finished in a small time window.

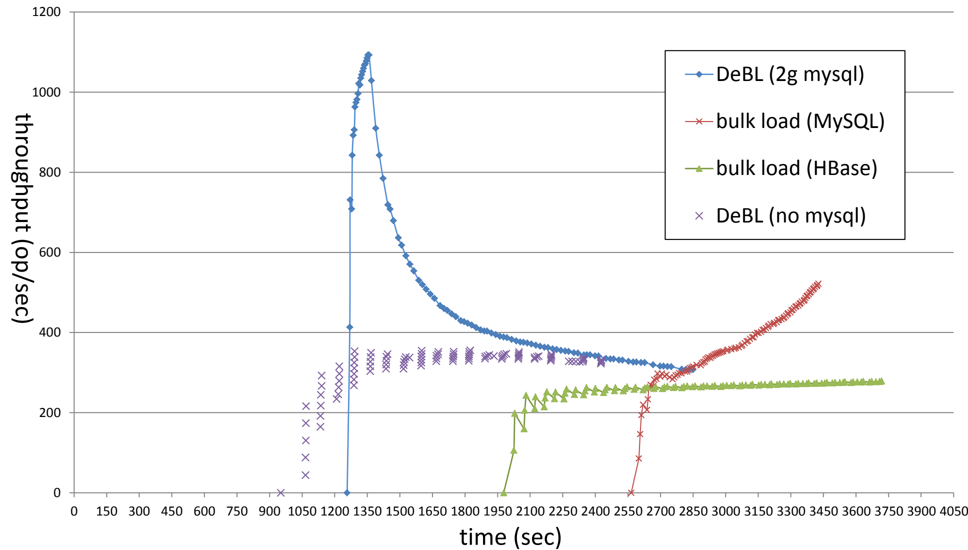


Figure 7.6: System Uptime & Query Throughput

In contrast to traditional bulk loading approaches, our DeBL approach trades complete data availability for fast system uptime. It provides incremental availability to files stored in HDFS on demand. It can be seen in the **DeBL (no mysql)** variant that the system started the earliest at 952.17s but the query throughputs occurred intermittently as relevant file partitions continuously got available in HBase. The throughputs were higher than **bulkLoad (HBase)** at the beginning since less available files needed to be scanned in HBase. Along with growing data size, the throughputs kept close to the highest value in **bulkLoad (HBase)** (332 op/sec). Using **DeBL (2g mysql)**, the system uptime had 300s delay whereas its throughput curve first climbed up drastically and reached its peak 1092.7 op/sec. After that, it began to fall and finally converged with **bulkLoad (HBase)**'s curve. The reason is that a big portion of frequently emerging queries were immediately answered by the 2GB hot data in MySQL at first. As the size of data in MySQL was much smaller, their query latencies were also faster than those on 10GB data in **bulkLoad (MySQL)**. The rest of the queries that could not be answered by MySQL were buffered by the coordinator and released as soon as the data was available in HBase. Import-

ant to mention, both DeBL variants were able to digest the entire 500,000 requests before the system uptime began in **bulkLoad (MySQL)**.

7.4 Summary

With hotness and coldness in real data distributions observed from social graphs, we exploited the trade-off between fast availability and low query latency and proposed our demand-driven bulk loading scheme. This scheme utilizes a hybrid storage platform consisting of a fast-load/slow-query HBase and a slow-load/fast-query MySQL to accommodate large-scale social graphs, which is a compromise with fast available "hot" graph data and slowly accessible "cold" data. Our experimental results show that our approach provides fast system uptime and incremental availability to "cold" data on demand.

Note that we do not assume that the data partition stored in MySQL is always hot since the "hotness" of files that reside in HBase can still be discovered in our approach by prioritizing the migration of more frequently queried data buckets. The limitation is that the query latency of HBase is not satisfactory if the data partition stored in HBase gets frequently accessed in the future. It is attractive to re-balance hot and cold data in dynamic workloads.

8 Real-time Materialized View in Hybrid Analytic Flow

In this chapter, we detail how "On-Demand ETL" is exploited in the *hybrid analytic flows* for next-generation operational business intelligence.

8.1 Motivation

Modern cloud infrastructure (e.g. Amazon EC2) aims at key features like performance, resiliency, and scalability. By utilizing NoSQL databases and public cloud infrastructure, business intelligence (BI) vendors are providing cost-effective tools in the cloud for users to gain more benefits from increasingly growing log or text files. Hadoop clusters are normally deployed to process large amounts of files. Derived information would be further combined with the results of processing operational data in traditional databases to reflect more valuable, real-time business trends and facts.

With the advent of next-generation operational BI, however, existing data integration pipelines cannot meet the increasing need of real-time analytics because complex Extract-Transfom-Load (ETL) jobs are usually time-consuming. Dayal et al. mentioned in [DCSW09] that more light-weight data integration pipelines are attractive which merge a back-end integration pipeline with a front-end query pipeline together as generic flows for operational BI. In [SWCD12], Simitsis et al. described their end-to-end optimization techniques for such generic flows to alleviate the impact of cumbersome ETL processes. Generic flows can be deployed to a cluster of execution engines and one logical flow operation can be executed in the most appropriate execution engine according to specific objectives, e.g. performance, freshness. These generic flows are referred to as *hybrid analytic flows*.

In traditional BI, offline ETL processes *push* and materialize operational source data into a dedicated warehouse where analytic queries do not have to compete with OLTP queries for resources. Therefore, query latency is low, whereas data is stale. In contrast, hybrid flows use data federation

techniques to *pull* operational data directly from data sources on demand for online, real-time analytics. However, due to on-the-fly executions of complex ETL jobs, analytic results might be delivered with poor response times. A trade-off exists here between query latency and data freshness.

Incremental ETL could be exploited instead of fully re-executing ETL jobs in hybrid flows. For some flow operations, for example, filter or projection, incremental implementations outperform full reloads. However, for certain operations (e.g. join), the costs of increment variants vary on metrics (e.g. the size of deltas) and sometimes could be worse than those of original implementations.

Therefore, we argue that it is not enough to compare the incremental maintenace costs for selecting materialized views at the warehouse [Gup97]. Usually, there is a *long way* (complex, time-consuming ETL jobs) from data sources through ETL processes to the warehouse. As ETL processes consist of different types of operations, incremental loading does not always perform stably. The location of the target tables (i.e. in data warehouses) has become a *restriction* on view maintenance at runtime using incremental loading techniques. Therefore, new thinking has emerged to decouple the materializations (in virtual/materialized integration) from their locations (data sources or the warehouse).

A *real-time materialized view* approach in hybrid flows is proposed to optimize performance and meanwhile guarantee data freshness using "On-Demand ETL". Depending on several metrics like source update rate and original/incremental operation cost from different platforms, we define a metric called *performance gain* for us to decide where certain materialization *points* would be in a flow and which flow operators are supposed to be executed in which way, i.e. either incrementally or with full-computing. Furthermore, we explore various platforms to accommodate derived materialized views.

8.2 Incremental Recomputation in Materialized Integration

Griffin et al. [GLT97] proposed delta rules based on relational algebra expressions to propagate the changes/deltas captured from base tables to materialized views. For example, the delta rules of a selection $\sigma_p(S)$ and a natural join $S \bowtie T$ are $\sigma_p(\Delta(S))$ and $[S_{new} \bowtie \Delta(T)] \cup [\Delta(S) \bowtie T_{new}]$, respectively (S, T denote two relations, Δ denotes change data). Given source deltas as input, new deltas are calculated at each operational level

in the view query plan until the final view deltas are derived (called incremental view maintenance). In data warehouse environments, warehouse tables are populated by (daily) ETL processes. Similar to maintaining materialized views incrementally, Griffin's delta rules can be also applied to certain logical representations (like Operator Hub Model [DHW+08]) of ETL processes to propagate only deltas from source tables to warehouse tables.

An incremental join variant is illustrated on the left side of Figure 8.1. As shown, to calculate the delta Δ_{JOIN} for an old result of table $orders$ joining table $lineitem$, new insertions Δ_O on table $orders$ are first joined with the new state of table $lineitem$, i.e., L_{new}, and further put together with the join result of Δ_L and O_{new} as a union. Afterwards, the rows of $\Delta_L \bowtie \Delta_O$ have to be subtracted from the union result to eliminate duplicates. We ran a small test to observe the latencies of joining $lineitem$ and $orders$ (scale factor 2) in a relational database rdb (Postgresql [pos]) and an ETL tool etl (Pentaho Kettle [CBVD10]) on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8 GB RAM, 1 TB SATA-II disk) using both incremental and original implementations.

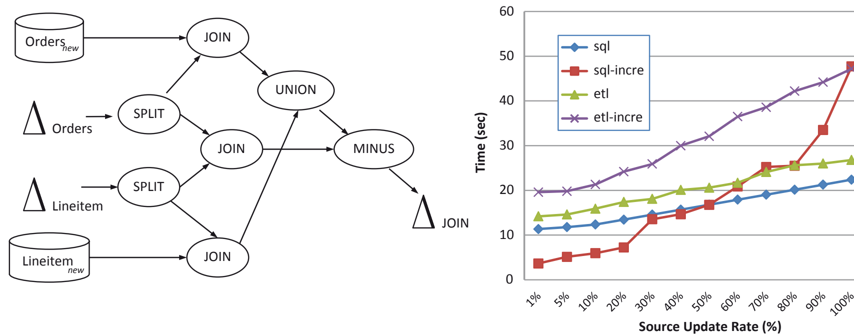


Figure 8.1: Incremental Join (*left*) and performance comparison between etl and rdb (*right*)

During the test, we steadily increased the size of delta inputs which simulates the source update rates.

We found (shown on the right side of Figure 8.1) that, in rdb, incremental join outperforms original join only with up to 50 % update rate (sql vs. $sql - incre$). With continuously increased update rates, the performance of incremental join degrades dramatically. We observed that when the delta input is small enough to fit into memory, an efficient hash join can be used with an in-memory hash table. If not, a slower sort-merge join instead will be used which contains an additional pre-sort step. The performance of two hash joins, each of which has a small join table (Δ < 50

%), can be more efficient than one sort-merge join on two big tables. However, with the same input size, performing two sort-merge joins ($\Delta > 50$ %) is definitely slower than one sort-merge join. Surprisingly, in the ETL case (etl vs. $etl - incre$), the performance of incremental join could never compete with the original one. Since the version of Pentaho Kettle we used does not support hash join, merge join was used which introduces additional sort stages. With limited support on join, the number of join dominates the overall performance and the performance of incremental join is always lower than the original one.

From the experimental results, we see that the performance of incremental recomputations depends on several metrics. Source update rates are important as they decide the size of input deltas which further affects the execution cost. In addition, platform support and execution capabilities play important roles as well. Furthermore, the delta rules of certain operations can also restrict efficient incremental implementations. In the next section, we will describe our real-time materialized view approach for hybrid flows based on a running example.

8.3 Real-time Materialized Views in Hybrid Flows

Throughout this chapter, we will use a running example to illustrate our real-time materialized view approach in hybrid flows proposed in the following. Consider the sample analytic flow depicted in Figure 8.2. It enables a business manager to check the state of incomplete orders in a specific branch, nation or region. Here he can prioritize the tasks with important metrics (e.g. the earliest order, the longest delay, total amount of revenues, and the number of orders) shown in the flow results and assign them to regional managers. In particular, real-time trends of sales and logistics can be captured by running analytic flows over source datasets and reactions will be immediately taken in case of emergency.

The sample flow extracts data from the orders table and the line item files and captures incomplete orders whose order date is earlier than the current date and at least one item of which has ship date later than the current date. One Lookup stage is first used to append customers' nation/region to the order stream. Line items are joined with orders after duplicate elimination and revenue calculation. Aggregation functions min, max, sum, count are then applied to the join results with grouping keys branch, nation, region. The insight is at last shown to a business manager with re-

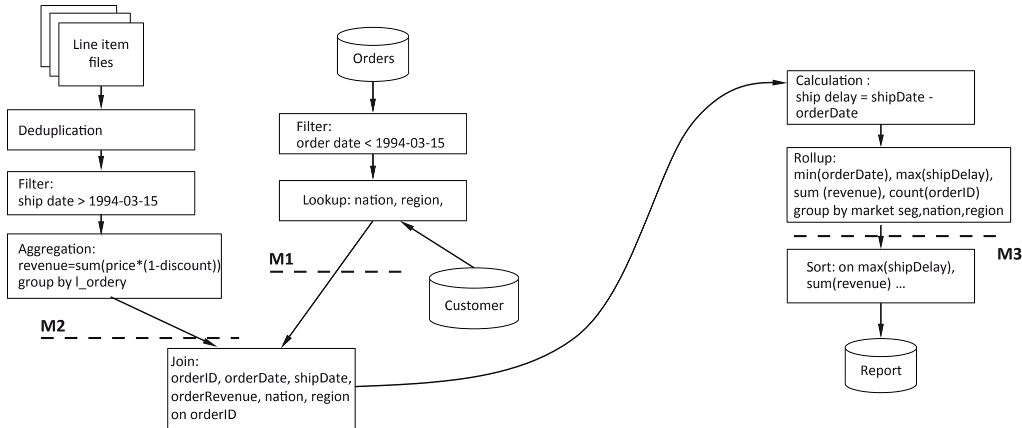


Figure 8.2: An example flow checking states of incomplete orders

sults sorted on important attributes (e.g. max(shipDelay)).

This logical flow describes the transformation and operation semantics of each stage. At flow runtime, multiple platforms can get involved. For example, large amounts of line item files with structured information can be loaded into a relational database where indexes and efficient processing are provided. They can be also quickly loaded into a distributed file system and processed by Hadoop jobs in parallel. Besides, the order table and the customer table may reside in different databases and an ETL tool is needed to combine information from multiple sources together. Depending on the overlap in functionality among different platforms, a single logical stage operation can be deployed to different engines. The challenge of optimizing analytic flows spanning multiple platforms (referred to as hybrid flows) has already been addressed in [SWCD12]. Optimizing approaches like data/function shipping can cut back to techniques used in distributed query processing [Kos00]. The main idea is to ship operation to the platform which has its most efficient implementation. If the required datasets are stored on another remote platform, datasets will be shipped from remote to local only on the condition that the local execution still outperforms the remote one with additional data moving cost. Otherwise, this operation will be executed remotely.

To keep the business manager informed of real-time insight on incomplete orders, this hybrid flow needs to be executed repeatedly. However, even if there are relatively small changes (e.g. insertions, updates, deletions) on the source side, unchanged data in source datasets still needs to get involved in the flow execution again to recompute new results. Such repeating work slows down the flow execution and meanwhile introdu-

ces large interferences and high loads to individual source systems. Furthermore, we observed that when executing flows on the Hadoop platform, in many cases, the output of a chain of MapReduce(MR)/Hadoop jobs that process a large amount of unstructured data is well structured and significantly smaller than the original input. Such an intermediate result is better processed in a relational database instead of in Hadoop.

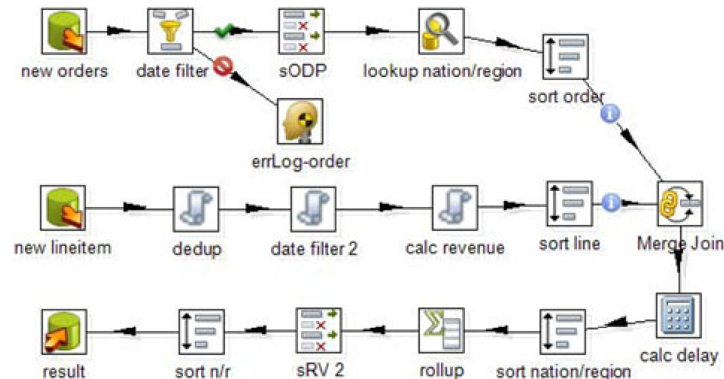


Figure 8.3: Original Flow

Many ad-hoc hybrid analytic flows overlap regarding flow operations, especially those in data integration pipelines which provide consolidated data. The observations above expose improvement potential of setting up materialized views in overlapping fragments of hybrid flows. To ensure the data consistency at flow runtime, which means providing the same view of sources as executing the flow against original datasets, view maintenance has to take place on demand. This process can benefit from incremental recomputations through which only the deltas on source datasets are taken as inputs. The logical hybrid flow discussed above is modelled using an ETL design tool and the original flow instance is shown in Figure 8.3. Each logical operation is mapped to a corresponding step supported by this tool. For example, join is explicitly implemented as sort merge join in the flow for fast processing. Note that this is just a representation of the logical flow and some flow fragments are still allowed to be deployed to more efficient platforms other than ETL tool.

To place a boundary in the flow for a materialized view, the cost of incremental implementations of flow operations will be taken into account. Recall that, as we showed in Section 8.2, incremental recomputations of specific operations (e.g. joins) sometimes perform worse than original execution and performance varies on different platforms. Assuming that the system is capable of capturing change data from sources,

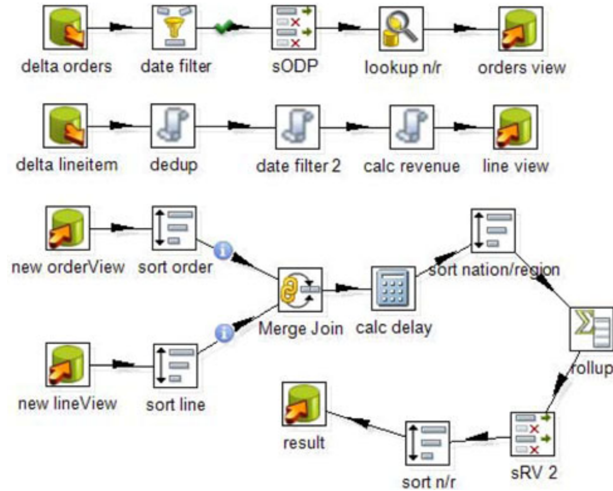


Figure 8.4: Optimized flow with materialized views before join

the flow instance shown in Figure 8.4 has two materialized views set up on the boundaries (M1, M2) in the logical flow in Figure 8.2. Intermediate results are materialized from previous execution of the two fragments and maintained with the deltas of the order table and the line item files in an incremental manner by the operations before the boundary at next runtime. Meanwhile, the subsequent join operator accesses the data from these two views for further execution using original join implementation. As we see from the experimental results in Section 8.2, the ETL tool etl can be used as the execution platform for this flow since the incremental join variant does not have advantage on etl, whereas another flow instance which has incremental join for maintaining view on the boundary M3 (see Figure 8.5) can only be deployed to the relational database rdb where the join performance can be guaranteed if the source update rates are lower than 50 %.

8.3.1 Performance Gain

With on-demand view maintenance at runtime, data freshness is guaranteed. In order to reduce the flow latency, the cost of each operator needs to be concerned into view selection step in terms of incremental and original execution. Let u_{org} denote the original operation u, u_{inc} its incremental variant and $c(u_t)$ the latency of each operator. u_{inc} is different from u_{org} in terms of the size of input (*delta* vs. *original*) and the implementation according to delta rules mentioned in Section 8.2. Thus, the latency difference $d(u)$ between incremental and original implementations of single

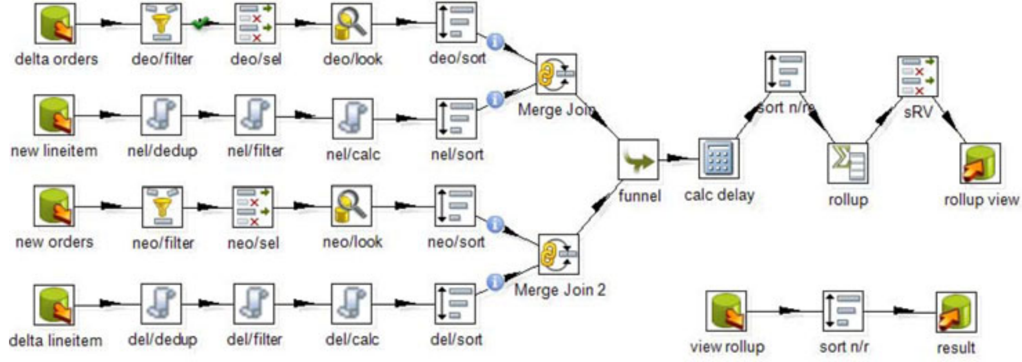


Figure 8.5: Optimized flow with materialized views after join

operator is given by the formula:

$$d(u) = c(u_{org}) - c(u_{inc}) \tag{8.1}$$

To simplify representing the latency of flow execution, here we only consider flows with operations running in sequence. For specific platforms providing parallelism for sub-flow execution, for example, executing two fragments of a join in parallel, the cost of parallelism here is the latency of the fragment with the maximum cost. The execution latency of original flow is the sum of latency of u_{org} as follows:

$$c(F_{org}) = \sum_{i=1}^{n} c_i(u_{org}) \tag{8.2}$$

With our approach, an original flow is cut into two fragments by a materialized view V . The preceding fragment consists of incremental operations and the following fragment has the remaining original operations. As we need the materialized view to be maintained at runtime, extra overhead $c_{online}(V)$ is introduced to load calculated deltas into the old view and read data from the new view for subsequent operations. Therefore, the latency of view-optimized flow is shown in the following:

$$c(F_V) = \sum_{i=1}^{m} c_i(u_{inc}) + c_{online}(V) + \sum_{i=m+1}^{n} c_i(u_{org}) \tag{8.3}$$

To quantify the benefits of materialized views on boundaries in hybrid flows, we define the performance gain $g(V)$ for each created view V. In the following formula, $c_{offline}(V)$ represents the cost of building up a materialized view offline, e.g. the storage waste on the platform accommodating this view. A weight σ is used here to make $c_{offline}(V)$ comparable

to flow latency. As one-off view generation could benefit multiple future hybrid flow, we put another variable λ as a weight to increase the view effect on the overall performance gain. Based on the Formulae 8.1, 8.2, 8.3 defined above, we give the formula of the performance gain of the materialized view as follows:

$$\begin{aligned} g(V) &= \frac{\lambda \times (c(F_{org}) - c(F_V)) - \sigma \times c_{offline}(V)}{c(F_{org})} \\ &= \frac{\lambda \times (\sum_{i=1}^{m} d_i(u) - c_{offline}(V)) - \sigma \times c_{offline}(V)}{c(F_{org})} \end{aligned} \tag{8.4}$$

Performance gain represents the rate of the improvement achieved from view-optimized flows as compared to the performance of the original flow. As the latency of the view-optimized flow increases, the performance gain decreases. In case F_V contains operations of incremental type which dramatically degrade the whole performance and lead to $c(Forg) \leq c(F_V)$, no performance gain exists. Furthermore, even if $c(F_{org}) - c(F_V)$ is positive, significant cost in creating views $c_{offline}(V)$ can also affect $g(V)$.

8.3.2 View Selection and Deployment

In this section, we will discuss how to implement our realtime materialized view approach in a hybrid flow. Two main challenges, i.e. view selection and view deployment, are involved. View selection denotes the process of finding out a position to insert a materialized view in the flow. This will be discussed only in a single-platform environment. In a cross-platform environment, a hybrid flow spans multiple platforms with different functionality. One problem left is which platform should be chosen to accommodate the view for the highest execution efficiency. This will be considered as view deployment problem.

View Selection. Figure 8.6 shows a selection table with a list of flow operations which are sorted according to their positions in the input flow.

To initialize this table, during previous flow execution, the latency of each operation with original implementation has been recorded as $c(u_{org})$. Meanwhile, the size of the intermediate result to be materialized is stored into $c_{offline}(V_i)$ as extra storage cost. Based on the change data capture mechanism, update rates on each source can be captured, which implies that $c(u_{inc})$ can be guessed using the size of given deltas. According to the formulae defined in the previous section, the latency difference $d(u)$ can be derived and further used to calculate the performance gain. In this example, symbols $+/-$ are simplified values for $d(u)$ instead of real values.

They indicate whether the difference is positive or negative and whether an incremental variant outperforms the original one or not. At last, the performance gain $g(V_i)$ for each flow operation u_i is calculated based on a simulated, view-optimized flow F_{V_i} where a materialized view V_i is supposed to be inserted right after operation u_i. All preceding operations (including u_i) before V_i would run incrementally and subsequent operations after u_i would be executed as u_{org}. Thus, $g(V_i)$ is the comparison result between the latency of this flow $(c(F_{V_i}))$ and that of original flow $(c(F_{org}))$.

Figure 8.6: View selection for a single-platform flow

After the initial phase, we consider the potential of optimizing performance with flow transition techniques. In [SVS05], Simitisis et al., introduced a set of logical transitions (e.g. swap, factorize, distribute, merge, split, etc.) that can be applied to ETL workflow optimization under certain condition. In this example, we only use the `SWAP` transition to interchange the sequence of adjacent operations in the flow. Starting from the first operation row in the selection table, we probe the values in the latency difference column $d(u)$ until we reach an operation row which has its $c(u_{inc})$ value as ∞ (u_6 in this example). This means that this operation is not able to support incremental recomputation, for example, sort operation and some user-defined functions. In the probe phase, u_6 is the first occurring operation which cannot be executed incrementally, which indicates that materialized view can only be set in front of it for executable incremental view maintenance. We record u_6 and probe further.

For the rest of the operations, we try to push the ones that have positive $d(u)$ in front of u_6 using multiple swap transitions since it is likely that certain operations (e.g. filter) could increase the performance gain. Two candidates u_7, u_8 have been found and u_8 failed as the movement

of u_8 cannot lead to an identical state of original flow. Until this step, we have a sequence of operations (u_{1-5}, u_7) as candidates of view boundary. Operation u_3 has the maximal performance gain 0.5 estimated and u_7 has only 0.2 since there are two operations u_4, u_5 sitting in front of it. Their incremental performance is lower than that of original implementations thus degrades u_7's gain. Therefore, we try to swap them with u_7 again and now u_7 achieves higher rank (due to the same state reason, u_7 cannot be pushed in front of u_4). The $g(V)$ of u_7 turns to 0.6 as the position of u_7 in the flow affect can affect the performance gain. After recomputing the performance gain, u_7 becomes the winner as materialization boundary in the flow, even if there is u_4 in front of it. This means u_7 brings significant benefit with its incremental variant, which offsets the negative impact from u_4. Using this method, we are able to find an appropriate boundary for materialized views in a single-platform flow.

View Deployment. Recall that the execution of hybrid flow spans multiple platforms with overlap in processing functionality. Logical flow operations can be deployed to different platforms for the highest efficiency. Introducing materialized views in hybrid flows and maintaining them on the fly can achieve additional improvement. For example, it may be effective for a Hadoop cluster to produce analytic results on terabytes of unstructured files as compared to a relational database. However, by caching small analytic results in the relational database, maintaining these views with smaller update files becomes much more efficient.

By extending the view selection table in a multiple platform environment, the cost of original and incremental operations will be compared across platforms. Those platforms that support incremental recomputations better are preferred for on-demand view maintenance. However, we still need to decide which platform should be used for storing the generated view data.

Suppose that a materialized view is inserted between two consecutive operations u_i and u_{i+1} (u_i run incrementally and u_{i+1} run with original impl.). If both operations are deployed to the same platform P_x , it is straightforward to materialize the intermediate results directly in this platform. If u_i runs on P_x and u_{i+1} on P_y, then we can maintain the view (i.e., compute the view deltas) on P_x and store the view onto P_y or move deltas onto P_y and maintain the view there. In order to answer this question, we extend the formula of *placement rate* defined in [SWCD12] as follows:

$$p(V) = \frac{\sigma \times c^{P_x}_{offline}(V) + \lambda \times (c^{i \to i+1}_{view} + c(u^{P_y}_{i+1}))}{\sigma \times c^{P_y}_{offline}(V) + \lambda \times (c^{i \to i+1}_{delta} + c(u^{P_y}_{i+1}))} \tag{8.5}$$

$c^{P}_{offline}(V)$ denotes the cost of initial loading of the materialized view in platform P. $c^{i \rightarrow i+1}_{view/delta}$ indicates whether to transfer the new views or the deltas onto platform P_y where u_{i+1} takes place. As we can see, if both platforms have no problem of initializing this view, it should be placed onto the platform P_y since the latency of transferring deltas is definitely lower. In case there is no more free space for view V on platform P_y, this view may be stored in another platform P_z where $c^{i \rightarrow i+1}_{delta} + c(u^{P_z}_{i+1})$ is lower than $c^{i \rightarrow i+1}_{view} + c(u^{P_y}_{i+1})$

8.4 Experiments

To validate our approach, we measured performance of three flow variants in our running example described in Section 8.3. This example has three flows: f_1 has flow operations running with their original implementations over source datasets; f_2 contains two materialized views ($M1$, $M2$) derived from previous execution results for the flow fragments before the join. The inputs are source change data captured in idle time and are used to maintain the two views at runtime; f_3 has a materialized view ($M3$) set after the join operation. The incremental join variant runs on the fly to calculate the deltas for the view.

Our experiments involve three data stores, namely orders, customer and lineitem files, and use the TPC-H benchmark. Experimental platforms are a relational database (rdb), an open source ETL tool (etl) and a M/R Hadoop engine (mr). The experiment consists of two parts. In the first part, flows f_1, f_2 and f_3 are tested on two single platforms rdb and etl, respectively. The goal is to prove that our view selection method can be used to find the best materialization points in transformation flows on a single platform. The second part takes flow f_2 as an example to show the benefits of our view deployment method. The flow fragment for processing lineitem files is either deployed to mr or first materialized and further maintained by rdb at runtime.

We ran our experiments in rdb and etl both running on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8 GB RAM, 1 TB SATA-II disk). Hadoop runs on a 6-node cluster (2 Quad-Core Intel Xeon Processor X3440, 4× 2.53 GHz, 4 GB RAM, 1 TB SATA-II disk, Gigabit Ethernet).

8.4.1 View Selection on Single Platform

We ran flows f_1, f_2 and f_3 in rdb and etl, respectively with input datasets of scale factor 1 (i.e. 1G). For f_2 and f_3, the update rates of orders and lineitems are 0.1, 1, 10, 30 and 50 %. Figure 8.7 shows the number of rows of input sources (orders, lineitems), derived views ($M1$, 2, 3) and deltas with 10% source update rate. The source table orders has 1.4M rows and lineitem files have 5.7M rows which are accessed directly by flow f_1. As compared to f_1, the number of the rows in the input ($M1$ + $M2$ + deltas) for flow f_2 is 1.6M and flow f_3 reads the smallest view ($M3$ + deltas, 60 K rows).

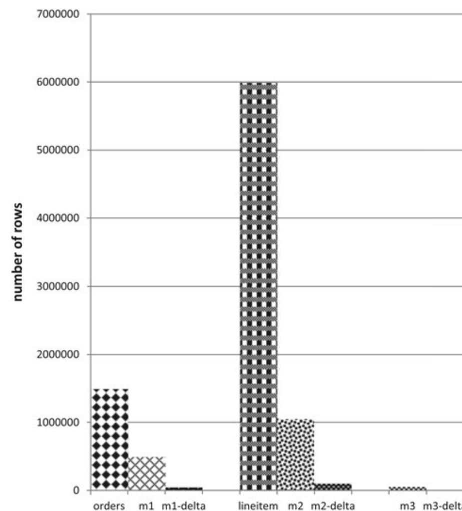


Figure 8.7: Data size overview

In Figure 8.8, there are 5 groups each of which represents the results of running 3 flows in rdb with different source update rate. In each group, the result of flow f_2 consists of three portions. The underlying two values represent the latency of maintaining and reading $M1, 2$ at runtime using incremental variants, respectively. The value on top is the runtime of running subsequent operations with original implementations after reading data out of $M1, 2$. Flow f_3 has only one view to maintain, thus consists of two values: maintenance time and execution time.

The results show that with increasing update rates, in rdb, flow f_2 always outperforms f_1. The incremental variants of the operations group, filter, and lookup that maintain $M1, 2$ in f_1 have lower latency than their original implementations, thus show high performance gain. Deltas can be calculated through each operator without accessing source data. In

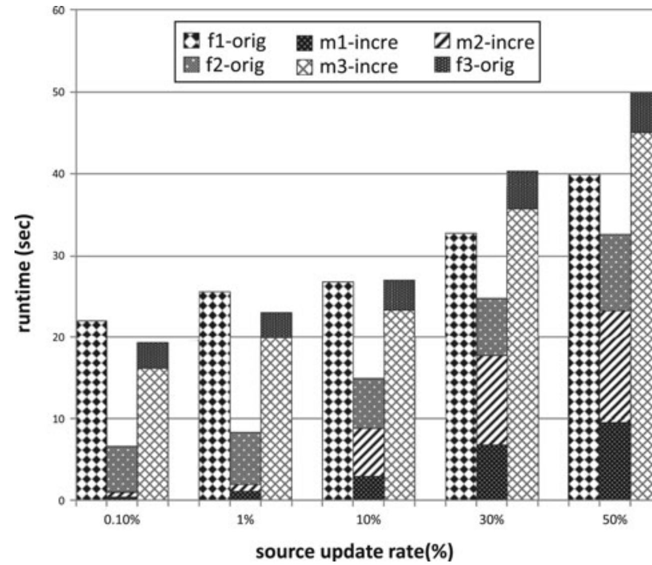


Figure 8.8: Running flows (f_1, f_2, f_3) in rdb

particular, as common group and filter operations have smaller output size than their input size, the size of the derived materialized views is much smaller. The impact of online delta-loading and view-reading is reduced. In contrast to f_2, flow f_3 shows worse performance and has higher latency than f_1 when the source update rate is higher than 10%. Even if f_3 has the smallest overhead for view-reading, the impact of the incremental join dominates the runtime since large volumes of source data need to be accessed for calculating deltas.

As shown in Figure 8.9, the results change in etl. The performance gain of views in f_2 can only be guaranteed with up to 30% update rate. Since etl does not have its own storage system, derived views are stored in rdb. The cost of moving data through the network highly reduces the performance gain.

In the next part of the experiments, we show that the performance varies from platform to platform. It depends not only on the operations supported by specific platforms but also on the source update rate. Using our view selection method, the performance gain of materialized views can be calculated in a cross-platform environment with different source update rates. Thereby, the best materialization points can be suggested in the flows.

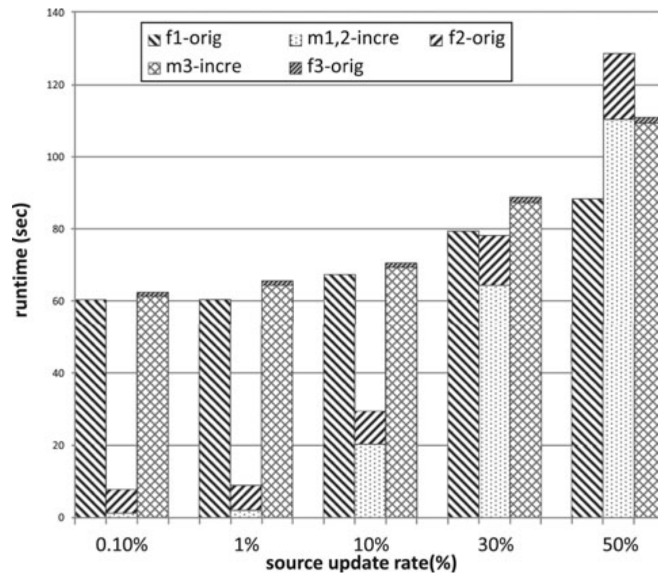


Figure 8.9: Running flows (f_1, f_2, f_3) in etl

8.4.2 View Deployment in Hybrid Flows

The second part validates the view deployment method in a hybrid flow. Data stores have scale factor 10 (i.e. 10G) in this part. Two variants of flow f_2 are used here. The first flow f_{2-1} has lineitem files that are stored in the Hadoop distributed file system (HDFS) of mr and processed by Hadoop jobs before join operation. The ETL tool etl joins the results from rdb and mr and performs the subsequent operations. In the second variant f_{2-2}, a materialized view V is created from the intermediate results of the Hadoop jobs and cached in rdb. At runtime, view V is maintained by first loading update data of the lineitem files into rdb and running the incremental variants of dedup, filter, and group to calculate deltas for V.

The results are depicted in Figure 8.10. The runtime of f_{2-1} is composed of the time of executing Hadoop jobs over lineitems and the time of running subsequent operations in etl. For f_{2-2}, the time is captured in load, maintenance and execution phases. With 0.1% update rate, the load phase and maintenace phase of f_{2-2} have lower latency as the delta size is small. Therefore, overall performance is significantly better than f_{2-1}. As the delta size increases to 10%, the performance of load operations in rdb degrades drastrically and the overall performance is worse than that of f_{2-1}.

The results give the suggestions that the overall performance can benefit from creating materialized views for the Hadoop jobs in rdb only if the

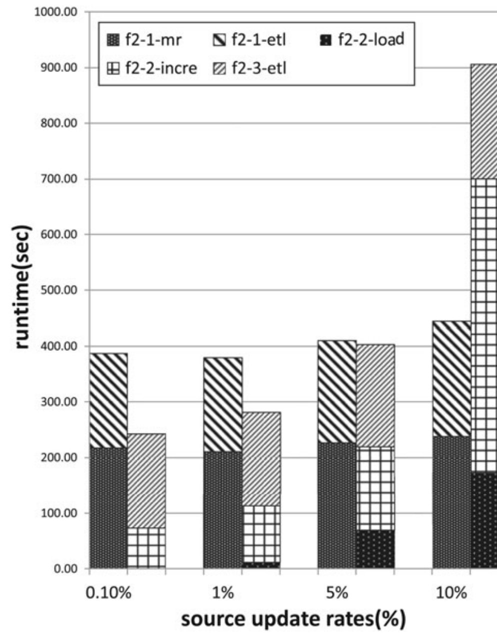


Figure 8.10: Running flow f_2 as a hybrid flow

lineitems contain update rates lower than 5%. This suits better in the case where f_{2-2} runs continuously. Otherwise, mr performs better if the delta size exceeds 5%.

8.5 Summary

In this chapter, we proposed a real-time materialized view approach for data flows which span multiple platforms like Hadoop, ETL and relational databases.

We first defined a metric called performance gain to identify the benefit of a materialized view in any possible point of a flow. Based on the performance gain, the flow operations that can achieve efficient incremental execution performance are pushed near sources using view selection method. The operations that have worse performance or not even support incremental processing are pushed after materialized views and are executed with original implementations. This solution works better in a heterogeneous environment where ETL flows extract data from sources with different update rates. With varying source update rates, the materialization point varies in the flow dynamically.

Furthermore, by using view deployment method, derived views can be

deployed to appropriate platforms for efficient maintenance at runtime. This method exploits the platforms with different support on incremental recomputations in hybrid flows.

The results showed that our real-time materialized view approach is able to give suggestions of using materialized views in a sequential flow execution environment where data consistency is guaranteed.

9 Conclusion and Future Work

In this chapter, we summarize the conclusions derived from all the previous chapters in this dissertation and give an outlook on interesting and challenging scenarios that could be explored in the future.

9.1 Conclusion

On-demand ETL can be considered as a special type of lazy materialized view maintenance. Both data warehouse tables and replicated tables in federated systems belong to one kind of materialization. The "On-Demand" thinking enables systems to better react to unpredictable, changing workloads. The main challenge is how to deliver the same performance as "Eager Maintenance" using sophisticated techniques, which turns *unpredictable* to *predictable*.

In this dissertation, we start with our consistency model which defines how an ETL job should be to bring an incoming query to a consistent state. Around this consistency model, we first proposed our incremental ETL pipeline in a standalone mode and showed that it can deliver processing performance close to real-time processing while still guaranteeing consistency. We then addressed this consistency model in a distributed environment. A distributed, incremental ETL pipeline is introduced which exploits partition parallelism to reduce the maintenance overhead and increase the request throughput. Furthermore, we addressed the elasticity property for incremental ETL pipeline and tackled the bottleneck operators by replacing in-memory queues with scalable HBase tables. A prototype implementation proved that it offers strict performance guarantees, which enables unpredictable queries to return in predictable times.

Apart from on-demand transformations, we simplified the consistency model and studied the problem concerning making the extraction phase be aware of varying freshness and execution deadlines. With overall performance gain analyzed, our workload-aware extraction processes could balance QoS for both OLTP and OLAP workload. Similar thinking was also exploited in the loading phase, which loads data on demand and achieves fast availability by sacrificing query latencies on cold data. In a

nutshell, we examined the "On-Demand" philosophy throughout all the phases of ETL and demonstrated its effectiveness and efficiency for mixed workloads with varying SLAs.

9.2 Outlook

With the trend of unifying domain-specific applications and infrastructures for *smart* business, a lot of interesting and challenging scenarios are yet to be explored.

Cloud Computing

On-demand ETL matches exactly the *pay-as-you-go* thinking in cloud computing. For services that require data with specific freshness needs (e.g., Data-as-a-service), on-demand ETL is able to deliver results with expected freshness using exact resources, i.e., pay only the resources as needed. Streaming ETL does not fit in this scenario, as it continuously delivers real-time stream data without taking the important *cost* aspect into consideration. Even though the data is most up-to-date, the freshness returned might be way beyond expectation, thus leading to extra costs.

Federated Computing

As introduced in Section 2.4.2, the BigDAWG system is a new-generation federated system which unifies domain-specific engines and infrastructures to handle complex applications. Processing engines like traditional relational databases, scientific databases, stream databases, MapReduce-based platforms, etc. are connected through a global mediator. We denote this scenario as *federated computing*.

In principle, users' programs can be executed over multiple engines that share common operators with the same processing functionality while the mediator schedules the program fragments based on one or more metrics, e.g., data locality. Streaming ETL can be used in this case to migrate relevant data to appropriate execution engines for maximal performance and data locality. However, due to the variety of applications supported by the BigDAWG system, queries can have different latencies and freshness. Therefore, we see a more suitable scenario here for on-demand ETL as only required data is transferred through network for processing.

Hybrid Flow

In Chapter 8, we examined improvement potential in *hybrid flows* by using real-time materialized views. The key behind this work is to break the traditional thinking of data warehousing and to apply view selection algorithms to more general, hybrid flows. On-demand ETL helps to deliver constant maintenance costs as required data is used to maintain the materialized views accessed by the ad-hoc queries, which simplifies the view selection process.

List of Figures

1.1 The scope of work in this dissertation 5

2.1 Architecture for Right-time ETL [TPL08] 18
2.2 Aurora Runtime Engine [CÇR+03] 22
2.3 Borealis Architecture [AAB+05] 23
2.4 High Level Architecture of Storm [TTS+14] 24
2.5 Spark Runtime [ZCD+12] 25
2.6 Lineage Graph for D-Streams and RDDs [ZDL+13] 26
2.7 Flink Process Model [CKE+15] 27
2.8 Flink Dataflow Graph Example [CKE+15] 28
2.9 S-Store Architecture [MTZ+15] 29
2.10 HBase Region Structure 33

3.1 Dataflow Graph 38
3.2 Incremental ETL Pipeline 41
3.3 Consistency Model Example 42
3.4 Scheduling Sink Transactions and OLAP Queries 45
3.5 Anomaly Example for Pipelined Incremental Join 47
3.6 Pipelined Incremental Join with Consistency Zones 48
3.7 Anomaly Example for ETL Pipeline Execution without Coordination 50
3.8 Pipelined SCD with Consistency Zone 52
3.9 Throughput and Latency Comparison 55
3.10 Average Latencies of 10 Ad-hoc Query Types in Read-heavy Workload 56
3.11 Average Latencies of 10 Ad-hoc Query Types in Update-heavy Workload 57
3.12 Job Latency Comparison for Consistency-Zone-aware MC 58

4.1 ETL Maintenance Flows for Distributed Databases 62
4.2 HBelt Architecture 64
4.3 Consistency Model in Distributed Environment 66
4.4 Distributed Snapshot Maintenance using Global/Local State Tables 69

4.5 Test Maintenance Flow in Kettle 71
4.6 Request Throughput after Issuing 1000 Requests using Diverse Delta Sizes 72
4.7 Request Throughput with Small Deltas (10K purchases & 100K lineitems) 73
4.8 Request Throughput with Large Deltas (50K purchases & 500K lineitems) 73

5.1 Embedding Elastic Queue Middleware in Streaming Processing Engine 78
5.2 EQM-specific Partitions in HBase 80
5.3 Enqueue Latency with and without Compactions in HBase 83
5.4 Correlation between Cluster Throughput and MAPD around the mean for Throughput 86
5.5 Load-balancing Quality in HBaqueue and HBase 87
5.6 Load Distribution in HBase and HBaqueue 88
5.7 Batch-Enqueueing Latency in HBaqueue and HBase 89
5.8 Latency Distribution of Batch-Enqueueing 89
5.9 Batch-Dequeueing Latency in HBaqueue and HBase 90

6.1 QoT vs. QoA using Trigger-based CDC 99
6.2 System Architecture for Workload-aware CDC 101
6.3 QoT & QoA of Trigger-based CDC Variants 105
6.4 QoT & QoA of Log-based CDC Variants 106
6.5 QoT & QoA of Timestamp-based CDC Variants 106

7.1 Architecture of Demand-driven Bulk Loading 112
7.2 Execution Pipelines in the HDFS loader 113
7.3 Offline & Online MapReduce Jobs 114
7.4 Elapsed Time during Loading Phase 117
7.5 Detailed Latency in Loading Phase 118
7.6 System Uptime & Query Throughput 119

8.1 Incremental Join (*left*) and performance comparison between etl and rdb (*right*) 123
8.2 An example flow checking states of incomplete orders . . . 125
8.3 Original Flow 126
8.4 Optimized flow with materialized views before join 127
8.5 Optimized flow with materialized views after join 128
8.6 View selection for a single-platform flow 130
8.7 Data size overview 133
8.8 Running flows (f_1, f_2, f_3) in rdb 134

8.9 Running flows (f_1, f_2, f_3) in etl 135
8.10 Running flow f_2 as a hybrid flow 136

List of Tables

6.1 Symbol Semantics 96
6.2 Transactions in OLTP Source System 98
6.3 Transactions arrived in Data Staging Area 98
6.4 CDC Requests arrived in CDC 98

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the Borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[ACÇ+03] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[APBC13] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[BBD+15] Ute Baumbach, Patric Becker, Uwe Denneler, Eberhard Hechler, Wolfgang Hengstler, Steffen Knoll, Frank Neumann, Guenter Georg Schoellmann, Khadija Souissi, Timm Zimmermann, et al. *Accelerating Data Transformation with IBM DB2 Analytics Accelerator for z/OS*. IBM Redbooks, 2015.

[ben] BenchmarkSQL. https://sourceforge.net/projects/benchmarksql/.

[BJ10] Andreas Behrend and Thomas Jörg. Optimized incremental ETL jobs for maintaining data warehouses. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, pages 216–224. ACM, 2010.

[BLT86] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *ACM SIGMOD Record*, volume 15, pages 61–71. ACM, 1986.

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[CBB+13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[CBVD10] Matt Casters, Roland Bouman, and Jos Van Dongen. *Pentaho Kettle solutions: building open source ETL solutions with Pentaho Data Integration*. John Wiley & Sons, 2010.

[CÇR+03] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 838–849. VLDB Endowment, 2003.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CEMK+15] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. Implementation of multidimensional databases in column-oriented NoSQL systems. In *East European conference on advances in databases and information systems*, pages 79–91. Springer, 2015.

[CFE+15] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[CHS+95] Michael J Carey, Laura M Haas, Peter M Schwarz, Manish Arya, William F Cody, Ronald Fagin, Myron Flickner, Allen W Luniewski, Wayne Niblack, Dragutin Petkovic, et al. Towards heterogeneous multimedia information systems: The garlic approach. In *Proceedings RIDE-DOM'95. Fifth International Workshop on Research Issues in Data Engineering-Distributed Object Management*, pages 124–131. IEEE, 1995.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[Dan17] Dipesh Dangol. Evaluation of a workload-aware change data capture approach for real-time analytics. Master's thesis, University of Kaiserslautern, 2017.

[DBBK15] Khaled Dehdouh, Fadila Bentayeb, Omar Boussaid, and Nadia Kabachi. Using the column oriented NoSQL model for implementing big data warehouses. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 469. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015.

[DCSW09] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 1–11. Acm, 2009.

[Deh16] Khaled Dehdouh. Building OLAP cubes from columnar NoSQL data warehouses. In *International Conference on Model and Data Engineering*, pages 166–179. Springer, 2016.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DHW+08] Stefan Dessloch, Mauricio A Hernández, Ryan Wisnesky, Ahmed Radwan, and Jindan Zhou. Orchid: Integrating schema mapping and ETL. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1307–1316. IEEE, 2008.

[DMTZ17] Jiang Du, John Meehan, Nesime Tatbul, and Stan Zdonik. Towards dynamic data placement for polystore ingestion. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, page 2. ACM, 2017.

[EDS+15] Aaron Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. A demonstration of the BigDAWG polystore system. *Proceedings of the VLDB Endowment*, 8(12):1908–1911, 2015.

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[fli] Apache Flink. https://flink.apache.org/.

[Geo11] Lars George. *HBase: the definitive guide: random access to your planet-size data*. O'Reilly Media, Inc., 2011.

[GJ11] Lukasz Golab and Theodore Johnson. Consistency in a stream warehouse. In *CIDR*, volume 11, pages 114–122, 2011.

[GJS09] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1207–1210. IEEE, 2009.

[GJS11] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk. Scalable scheduling of updates in streaming data warehouses. *IEEE Transactions on knowledge and data engineering*, 24(6):1092–1105, 2011.

[GJSS09] Lukasz Golab, Theodore Johnson, J Spencer Seidel, and Vladislav Shkapenyuk. Stream warehousing with DataDepot. In *Proceedings of the 2009 ACM SIGMOD Internatio-*

nal Conference on Management of data, pages 847–854. ACM, 2009.

[GLR05] Hongfei Guo, Per-Åke Larson, and Raghu Ramakrishnan. Caching with good enough currency, consistency, and completeness. In *Proceedings of the 31st international conference on Very large data bases*, pages 457–468. VLDB Endowment, 2005.

[GLRG04] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: how to say good enough in SQL. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM, 2004.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[GM+95] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GPA+15] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment*, 8(12):1716–1727, 2015.

[Gre17] Nikolay Grechanov. A scalable queue implementation on top of hbase for streaming ETL. Master's thesis, University of Kaiserslautern, 2017.

[GSHW13] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2013.

[Gup97] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *International Conference on Database Theory*, pages 98–112. Springer, 1997.

[Han87] Eric N Hanson. *A performance analysis of view materialization strategies*, volume 16. ACM, 1987.

[inf] IBM InfoSphere DataStage. https://www.ibm.com/products/infosphere-datastage.

[JD08] Thomas Jörg and Stefan Deßloch. Towards generating ETL processes for incremental loading. In *Proceedings of the 2008 international symposium on Database engineering & applications*, pages 101–110. ACM, 2008.

[JD09a] Thomas Jörg and Stefan Dessloch. Formalizing ETL jobs for incremental loading of data warehouses. *Datenbanksysteme in Business, Technologie und Web (BTW)–13. Fachtagung des GI-Fachbereichs"Datenbanken und Informationssysteme"(DBIS)*, 2009.

[JD09b] Thomas Jörg and Stefan Dessloch. Near real-time data warehousing using state-of-the-art ETL tools. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 100–117. Springer, 2009.

[KC04] Ralph Kimball and Joe Caserta. *The data warehouse ETL toolkit*. John Wiley & Sons, 2004.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[KVP05] Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. ETL queues for active data warehousing. In *Proceedings of the 2nd international workshop on Information quality in information systems*, pages 28–39. ACM, 2005.

[KVS13] Anastasios Karagiannis, Panos Vassiliadis, and Alkis Simitsis. Scheduling strategies for efficient ETL execution. *Information Systems*, 38(6):927–945, 2013.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[MAZ+17] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. Data ingestion for the connected world. In *CIDR*, 2017.

[mic] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[MTZ+15] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. S-Store: streaming meets transaction processing. *Proceedings of the VLDB Endowment*, 8(13):2134–2145, 2015.

[mys] Mysql. https://www.mysql.com/.

[MZT+16] John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, and Aaron Elmore. Integrating real-time and batch processing in a polystore. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2016.

[ora14] *Best practices for real-time data warehousing*. Oracle white paper, 2014.

[pos] Postgresql. https://www.postgresql.org/.

[QBSD15] Weiping Qu, Vinanthi Basavaraj, Sahana Shankar, and Stefan Dessloch. Real-time snapshot maintenance with incremental ETL pipelines in data warehouses. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 217–228. Springer, 2015.

[QD14a] Weiping Qu and Stefan Dessloch. A demand-driven bulk loading scheme for large-scale social graphs. In *East European Conference on Advances in Databases and Information Systems*, pages 139–152. Springer, 2014.

[QD14b] Weiping Qu and Stefan Dessloch. A real-time materialized view approach for analytic flows in hybrid cloud environments. *Datenbank-Spektrum*, 14(2):97–106, 2014.

[QD17a] Weiping Qu and Stefan Dessloch. Distributed snapshot maintenance in wide-column NoSQL databases using partitioned incremental ETL pipelines. *Information Systems*, 70:48–58, 2017.

[QD17b] Weiping Qu and Stefan Deßloch. Incremental ETL pipeline scheduling for near real-time data warehouses. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[QD17c] Weiping Qu and Stefan Dessloch. A lightweight elastic queue middleware for distributed streaming pipeline. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 173–182. Springer, 2017.

[QD17d] Weiping Qu and Stefan Dessloch. On-demand snapshot maintenance in data warehouses using incremental ETL pipeline. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXII*, pages 91–112. Springer, 2017.

[QRD13] Weiping Qu, Michael Rappold, and Stefan Dessloch. Adaptive prejoin approach for performance optimization in MapReduce-based warehouses. *Grundlagen von Datenbanken*, 2013.

[QSGD15] Weiping Qu, Sahana Shankar, Sandy Ganza, and Stefan Dessloch. HBelt: Integrating an incremental ETL pipeline with a big data store for real-time analytics. In *East European Conference on Advances in Databases and Information Systems*, pages 123–137. Springer, 2015.

[Qu13] Weiping Qu. Global optimization in hybrid analytics system including MapReduce. PhD Consortium poster presentations at ADBIS, Genoa, Italy, 2013.

[R+11] Philip Russom et al. Big data analytics. *TDWI best practices report, fourth quarter*, 19(4):1–34, 2011.

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise

application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[RSQ+08] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *ICDE*, volume 8, pages 60–69, 2008.

[SAG+09] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[SC05] Michael Stonebraker and Ugur Cetintemel. "One size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*, pages 2–11. IEEE, 2005.

[spa] Apache Spark. https://spark.apache.org/.

[sto] Apache Storm. https://storm.apache.org/.

[SVS05] Alkis Simitsis, Panos Vassiliadis, and Timos Sellis. Optimizing ETL processes in data warehouses. In *21st International Conference on Data Engineering (ICDE'05)*, pages 564–575. IEEE, 2005.

[SWCD12] Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 829–840. ACM, 2012.

[SWDC10] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Malu Castellanos. Optimizing ETL workflows for fault-tolerance. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 385–396. IEEE, 2010.

[TFL09] Maik Thiele, Ulrike Fischer, and Wolfgang Lehner. Partition-based workload scheduling in living data warehouse environments. *Information Systems*, 34(4-5):382–399, 2009.

[tpca] TPC-C benchmark. http://www.tpc.org/tpcc/.

[tpcb] TPC-DS benchmark. http://www.tpc.org/tpcds/.

[TPL08] Christian Thomsen, Torben Bach Pedersen, and Wolfgang Lehner. RiTE: Providing on-demand data for right-time data warehousing. In *2008 IEEE 24th International Conference on Data Engineering*, pages 456–465. IEEE, 2008.

[tri] Trident tutorial. http://storm.apache.org/documentation/Trident-tutorial.html.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[TZM+15] Nesime Tatbul, Stan Zdonik, John Meehan, Cansu Aslantas, Michael Stonebraker, Kristin Tufte, Chris Giossi, and Hong Quach. Handling shared, mutable state in stream processing with correctness guarantees. *IEEE Data Eng. Bull.*, 38(4):94–104, 2015.

[VOE11] Richard L Villars, Carl W Olofson, and Matthew Eastwood. Big data: What it is and why you should care. *White Paper, IDC*, 14:1–14, 2011.

[VS09] Panos Vassiliadis and Alkis Simitsis. Near real time ETL. In *New Trends in Data Warehousing and Data Analysis*, pages 1–31. Springer, 2009.

[Whi12] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[WT15] Y. Wu and K. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International*

Conference on Data Engineering, pages 723–734, April 2015. doi:10.1109/ICDE.2015.7113328.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[ZE+11] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *ACM SIGMOD Record*, volume 24, pages 316–327. ACM, 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L Wiener. The Strobe algorithms for multi-source warehouse consistency. In *Fourth International Conference on Parallel and Distributed Information Systems*, pages 146–157. IEEE, 1996.

[ZLE07] Jingren Zhou, Per-Ake Larson, and Hicham G Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*, pages 231–242. VLDB Endowment, 2007.

Curriculum Vitae

Education

10/2013 - 3/2018	PhD candidate Heterogeneous Information Systems (HIS) Group University of Kaiserslautern, Germany
10/2006 - 4/2011	Diploma (Dipl.-Inform.) Department of Informatics Karlsruhe Institute of Technology (KIT), Germany
10/2002 - 7/2006	Bachelor of Engineering Department of Computer Science and Technology Tongji University, China

Work Experience

Since 4/2018	Software developer Ant Financial, Alibaba Group Shanghai, China
10/2011 - 3/2018	Scientific staff member Heterogeneous Information Systems (HIS) Group University of Kaiserslautern
1/2010 - 4/2011	Internship and diploma thesis student IBM Germany Research & Development GmbH Böblingen
4/2008 - 8/2011	Student assistant FZI Forschungszentrum Informatik Karlsruhe