Optimizing Multi-Way Joins for Adaptive, Scale-out Stream Processing

Dissertation

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

von

Manuel Dossinger

Datum der wissenschaftlichen Aussprache	14.12.2022
Dekan	Prof. Dr. Jens Schmitt
Gutachter	Prof. Dr. Sebastian Michel
	Prof. Dr. Mirek Riedewald

DE-386

Abstract

Processing data streams is a classical and ubiquitous problem. A query is registered against a potentially endless data stream and continuously delivers results as tuples stream in. Modern stream processing systems allow users to express queries in different ways. However, when a query involves joins between multiple input streams, the order of these joins is not transparently optimized. In this thesis, we explore ways to optimize multi-way theta joins, where the join predicates are not limited to equality and multiple inputs are referenced. We put forward a novel operator, MultiStream, which joins multiple input streams using iterative probing and bringing minimal materialization effort in. The order in which tuples are sent inside a MultiStream operator is optimized using a cost-based model. Further, a query can be answered using an multi-way tree comprising multiple MultiStream operators where each inner operator represents a materialized intermediate result. We integrate equi-joins in MultiStream to reduce communication, such that mixed queries of theta and equality predicates are supported. Streaming queries are long-standing and thus multiple queries might be registered at the system at the same time. Hence, we research joint answering of multiple multi-way join queries and optimize the global ordering using integer linear programming. All these approaches are implemented in CLASH, a system for generating Apache Storm topologies including runtime components that enables users to pose queries in a declarative way and let the system craft the suitable topology.

Zusammenfassung

Datenstromverarbeitung ist ein klassisches und universelles Problem. Datenströme sind endlos und das Ergebnis einer Datenstromanfrage ist wiederum ein Strom, daher ist eine Anfrage potentiell für eine unbegrenzte Zeit aktiv. Moderne datenstromverarbeitende Systeme erlauben es Benutzern, Anfragen auf verschiedene Arten auszudrücken. Allerdings wird die Reihenfolge von mehreren Join-Operationen nicht transparent optimiert, wenn eine Anfrage mehrere Eingabeströme involviert. In dieser Arbeit erforschen wir wie Mehrwege-Theta-Joins – das bedeutet, Prädikate sind nicht auf Gleichheit beschränkt und mehrere Eingaben werden referenziert – optimiert werden können. Wir stellen einen neuartigen Operator vor, MultiStream, welcher mehrere Eingabeströme mittels iterativen Sondierens verbinden kann und der minimalen Materialisierungsaufwand benötigt. Die Reihenfolge, in der Tupel innerhalb eines MultiStream-Operators gesendet werden, wird mit Hilfe eines kostenbasierten Models optimiert. Weiterhin kann eine Anfrage durch einen Mehrwegebaum beantwortet werden, welche aus mehreren MultiStream-Operatoren besteht und wobei jeder innere Operator einem materialisierten Zwischenergebnis entspricht. Um den Kommunikationsaufwand für Anfragen, die zum einen Teil aus beliebigen Prädikaten und zum anderen Teil aus Gleichheiten bestehen, zu verringern, integrieren wir Equijoinberechnung in MultiStream. Datenstromanfragen sind für längere Zeit aktiv and so können mehrere Anfragen gleichzeitig in einem System registriert sein. Für solche Fälle untersuchen wir das gemeinsame Beantworten mehrerer solcher Mehrwege-Join-Anfragen und optimieren die globale Reihenfolge mittels ganzzahliger linerarer Programmierung. Alle diese Ansätze haben wir in CLASH implementiert, ein System, das Apache-Storm-Topologien und passende Laufzeitkomponenten erstellt. Dieses System erlaubt Benutzern beliebige Anfragen deklarativ zu stellen und erstellt daraufhin eine passende Topologie.

Contents

Chapter 1

Introduction

Processing data streams is a classical and ubiquitous problem. It ranges from monitoring enterprise-internal system access logs for ad placement or anomaly detection, to providing real-time analytics over social network streams, from gathering distributed sensor data in a smart grid to algorithmic trading in electronic stock exchanges. Prominent engines like Spark Streaming [10], Flink [17, 8], Apache Storm's Trident [1], or Kafka [62, 9] allow users or higher-level applications to express queries in SQL-style declarative languages, and deploy and execute query plans over potentially very many compute tasks in a data center.

In most cases, such queries do not merely filter or aggregate tuples from a single relation but involve *joins* that connect information pieces from various sources. For instance, search-engine queries and ad-clicks need to be joined for billing purposes [7] and in complex event processing, events are commonly expressed by multiple criteria that do not originate from a single sensor [43].

Status Quo and Motivation

Database systems allow their users to express nearly arbitrarily complex queries in form of the declarative language SQL. These queries are fed into an optimizer that produces a plan that dictates the underlying data engine in which order which operations are to be performed such that a result can be sent back to the user. This way, the query and the execution plan are decoupled from each other, and the same query can be answered using different plans depending on, e.g., properties of the underlying data or available resources. In particular, these queries can involve user-defined predicates and an arbitrary number of joins.

Prevailing scale-out stream processing systems, on the other hand, often support a SQL-like declarative language, but only a narrow band of queries. E.g., Flink and Kafka only provide equi joins, where the syntactic order of the relations implies the sequence of the join operators, and Storm's SQL implementation does not support joins at all. Thus, if users need the system to compute a (non-equi-)join query, they must be proficient in writing the query in the programmatic interfaces of these systems.

In a truly declarative interface, there should be no restriction on the predicates or order of the input relations in a multi-way join query. The underlying system should understand which relations need to be joined and craft an execution strategy which yields the correct result with as little effort as possible. Flink supports multiple join

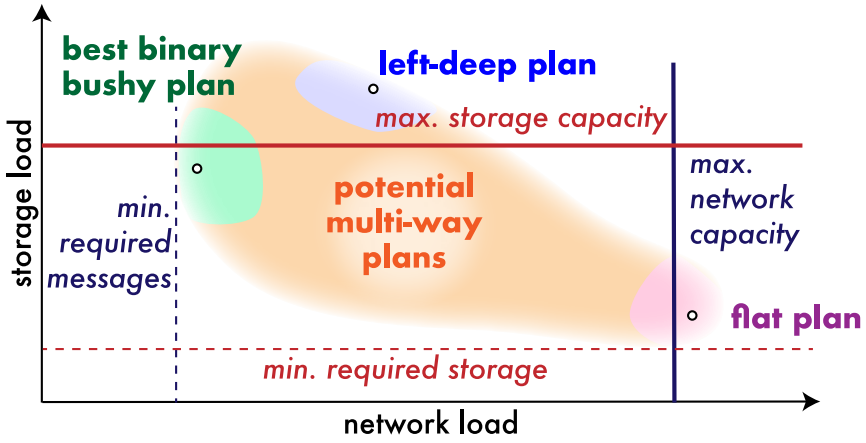


Figure 1.1: The space of different types of join plans regarding storage and network load.

relations in a SQL query, however, the order in which these relations are listed in the query string prescribes the order of the join operations in the final query execution plan. This blind spot for multi-way theta joins motivates this work of research where we explore what is necessary to implement theta-join ordering strategies for scale-out architectures and implement prototypes to answer joins, not limited to two inputs or equality predicates.

Our Approach

The wide spectrum of join algorithms is illustrated in Figure 1.1, regarding required storage for materializing the join state and network usage. The amount of memory available in the compute cluster executing the query gives us the maximum storage capacity, and the interconnection of nodes induces the maximum network capacity. Query plans that exceed the available storage or network bandwidth are infeasible. On the other hand, the query and the data themselves impose minimal requirements, e.g., each tuple that might still be involved in a future join result needs to be stored, and each fresh arriving tuple needs to be sent somewhere for probing. The position of a join plan in this graph depends on many factors like the actual query, data characteristics, configuration of the involved nodes, or local algorithms used. A query plan with a fixed structure like a left-deep plan composed of binary operators might require more storage for intermediate results than available, and a flat plan that relies on iterative probing through all base relations might overflow the network's capacity. However, there are various other possible plans between these two extremes, like a bushy, binary plan which materializes less intermediate results than the left-deep

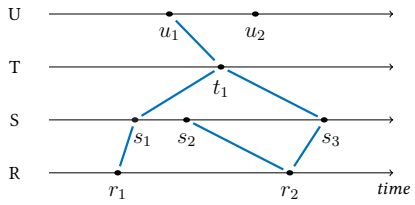


Figure 1.2: Tuples of multiple input streams arrive.

plan. Further, bushy join plans consisting of multi-way joins often require even less storage at the cost of higher network utilization.

1.1 Problem Statement

Given streamed relations $R_1, \ldots, R_m$, each representing a continuously arriving sequence of tuples, we are interested in the join between $R_1, \ldots, R_m$, such that $r_1 \circ \cdots \circ r_m$ (the concatenation of these tuples) with $r_i \in R_i$ is output iff a predicate $\theta(r_1, \ldots, r_m)$ is satisfied. This output should be produced as soon as the latest tuple of $r_1, \ldots, r_m$ arrives. The join predicate θ is a conjunction of binary predicates over two relations, i.e., $\theta = \bigwedge_{1 \leq i,j \leq m, i \neq j} \theta_{i,j}$. The binary predicates are not further restricted, e.g., $\theta_{1,2}$ could filter all tuples where attribute a in relation R_1 is similar to attribute b in relation R_2 for a given similarity definition, or $\theta_{2,3}$ could be satisfied if a combination of attributes of the R_2 tuples adds up to the same number as a single attribute in R_3. We assume nothing about the predicates but decidability for Chapter 5, and in Chapter 6 we exploit queries where some $\theta_{i,j}$ are equality predicates.

For such a join query we want to produce a computation strategy which ensures correctness and is efficient for given assumptions about the amount of data arriving at the system and produced as intermediate join results. This produced strategy is deployed to a distributed and parallel system which consists of stateful components that can communicate with each other. State is kept in main memory and the processing and communication strategy needs to make sure that the query is answered correctly and join results are provided in time. The state of materialized inputs and optional intermediate results can be distributed over multiple nodes of a compute cluster, however, each tuple is at exactly one location, i.e., there is no data redundancy. The ideas we provide are oriented at Apache Storm which serves as least common denominator for stream processing frameworks and, thus, are applicable to other systems as well.

Further, the query does not provide hints for how to produce the result, but the user can formulate a query in a purely declarative way, and the system is responsible for choosing a strategy. While this is the standard for (SQL-)database systems, modern streaming systems do not normally support this. For example, the Apache Flink doc-

umentation explicitly states, that it does not optimize the join order [2]. This means, if a user writes a query with the from clause `FROM R, S, T`, the execution engine will first compute the join between `R` and `S`, and then join this result with `T`.

Examples

Consider the example depicted in Figure 1.2, where relations R, S, T, and U are joined with $\theta = \theta_{R,S} \wedge \theta_{S,T} \wedge \theta_{T,U}$. Time is evolving from left to right, so the first tuple that arrives at the system is r_1. Two tuples connected by a blue line satisfy the partial join predicate, e.g., r_2 and s_2 satisfy $\theta_{R,S}$ and thus are connected. Such tuples *might* belong to the overall join result. However, since s_2 does not find a join partner in T (yet), they do not belong to the result (yet). On the other hand, r_1, s_1, t_1, and u_1 simultaneously satisfy all partial predicates, thus, should be joined and included in the result, likewise for r_2, s_3, t_1, and u_1. This example underpins that tuples, once observed, have to be stored (e.g., in main memory) as they might be join partners for tuples that are arriving later-on. Also, partial joins can be stored (e.g., $u_1 \circ t_1$) such that a later arriving tuple can produce the overall result more quickly. This abstract processing strategy needs to be executed by a distributed system. This means, when tuple s_3 arrives the system needs to make sure, it meets t_1 and r_2, as these tuples satisfy the predicate. Because s_3 and r_1 do not satisfy the predicate they do not need to (but can) be at the same distributed processing node.

For a more practical example, consider different streams of social media postings. A person analyzing patterns in social media might be interested in messages referencing the same newspaper headline that appears first on some network A, then spreads to networks B and C. So, the query would join the streams of A, B, and C, where each tuple has an attribute τ indicating the timestamp of the posting and m for the message, and another stream N of newspaper articles with attribute h for the headline. The join predicates are $A.\tau \leq B.\tau$, $A.\tau \leq C.\tau$ to make sure that the message of A comes first. For the messages to reference the same headline, $N.h$ needs to be contained in $A.m$, $B.m$, and $C.m$. Due to the nature of social networks, the matching should be fuzzy to account for spelling errors or corrections, which is done by predicate θ. Thus, the predicates $\theta(N.h, A.m)$, $\theta(N.h, B.m)$, and $\theta(N.h, C.m)$ are also added to the overall predicate. The person only needs to formulate this query in terms of the join predicates, but is not concerned with where in a distributed system which tuples reside and where they are sent in order to evaluate which predicates.

1.1.1 Notation and Nomenclature

Throughout this thesis we will use the notation displayed in Table 1.1.

We denote the input datasets for join queries as **(streamed) relation**. These relations contain a sequence of tuples and each tuple consists of a timestamp as well as

Table 1.1: Notation used in this thesis.

Notation	Description
R, S, T, R_i	Relations
r, s, t	Tuples of according relations
N_R	Number of partitions of relation R, also number of tasks handling relation R
N	Number of partitions/tasks in total
$\|R\|$	Size of relation R
$\theta_{R,S}$ $(\theta_{i,j})$	Join predicate for R and S (R_i and R_j)
$f_{R,S}$ $(f_{i,j})$	Selectivity of R and S (R_i and R_j)

attribute values. We assume, that all tuples of a relation have an attribute value if it is referenced by a join predicate. Further terms will be introduced where appropriate.

A **query** can be seen as a relation that contains the tuples of the query result. For example, a query q that joins two relations R and S can be written as $q := \{r \circ s \mid r \in R, s \in S, \theta(r, s)\}$. Most of the time, we are only interested in the involved relations and the join predicates. Thus a query is a set of relation identifiers and predicate expressions, e.g. $q_1 = (R, S, R.a = S.b)$ for a binary join, $q_2 = (R, S, T)$ for a ternary cross product, or $q_3 = (R, S, T, \theta_{R,S}, \theta_{R,T}, \theta_{S,T})$ for a ternary join with variables indicating predicates between all pairs of the input relations. Later in the thesis, we will add window constraints to the relation identifiers, and write them following the work on CQL by Arasu et al. [11] in brackets, e.g. $q_4 = (R[\infty], S[1h], \theta_{R,S})$. An omitted window constraint can be interpreted as $[\infty]$. Sometimes, a tuple centric view on a join is required, then we write $r \bowtie S$, where r is a single tuple and S is a relation.

1.2 Contributions and Publications

The contributions of this line of work can be summed up as follows:

- We introduce a novel distributed and scalable operator for computing n-way joins in a memory efficient way, the MultiStream operator.
- We explain the composability of n-way operators into n-ary trees as basis for optimization of multi-way join processing in streaming systems and propose a cost-based optimization framework for this optimization
- With this, we build the basis for declarative multi-way join querying in data streams, which to our knowledge is not supported by contemporary stream processing systems.

- We implement our proposed strategies into a system, CLASH, that can indeed handle arbitrary input, proving the versatility of our approach.
- We show how to use integer linear programming as vehicle for optimizing multi-query stream processing plans.

The work in this paper was submitted and accepted by the following peer-reviewed conferences and workshops.

First, we put forward the initial work for the MultiStream operator that allows creation of n-ary operator trees in the BeyondMR workshop 2017 [35]. After that, we presented the full-fledged optimization approach including cost-models and optimization algorithms in BigData 2019 [24].

- Manuel Hoffmann and Sebastian Michel. Scaling Out Continuous Multi-Way Theta Joins. Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR), Chicago, IL, USA, 2017, co-located with SIGMOD/PODS [35].
- Manuel Dossinger and Sebastian Michel. Scaling Out Multi-Way Stream Joins using Optimized, Iterative Probing. IEEE BigData 2019 Conference, Los Angeles, CA, USA [24].

We modified the approach to allow joint optimization and execution of multiple queries, as well as adaptive re-optimization, and presented this in ICDE 2021 [25].

- Manuel Dossinger and Sebastian Michel. Optimizing Multiple Multi-way Stream Joins. 37th IEEE International Conference on Data Engineering (ICDE) 2021, Chania, Greece [25].

Due to its fundamental nature, CLASH is part of all publications, but most significantly the system was presented on a Demo at SIGMOD 2019 [26].

- Manuel Dossinger, Constantin Roudsarabi, and Sebastian Michel. CLASH: A High-Level Abstraction for Optimized, Multi-Way Stream Joins over Apache Storm. Demo on SIGMOD 2019, Amsterdam, Netherlands [26].

1.3 Outline

Chapter 2 introduces the background of both, distributed join and stream computation, the ideational predecessor of MultiStream, the BiStream operator [47], as well as a reference for the notation used in this thesis. Chapter 3 relates this line of work to the other active and prior research of join computation.

Chapter 4 provides first insights into our research-prototype, CLASH. Here, we give a high-level overview which is necessary to understand the following Chapters.

The MultiStream operator, explained in detail in Chapter 5, is the basis for our approach of computing multi-way stream joins. We present a detailed model to quantify the storage requirements for materialization of intermediate results and number of messages sent, respectively. We compose multiple MultiStream operators into operator trees and introduce optimization strategies for finding trees that yield efficient processing strategies.

Chapter 6 contains our extensions to the MultiStream approach to incorporate windows and equi joins. The former restrict the potential join partners to only those tuples that are at most a window length apart, thereby enabling eviction of old tuples and freeing resources. The latter allow more efficient routing strategies, reducing overall computation load.

Chapter 7 breaks with the single query view and introduces our approach to multi join-query optimization using integer linear programming to find combined routing strategies. In this chapter, we propose to allow reacting to changes in the underlying data characteristics to avoid eventually running an unoptimized strategy.

In Chapter 8, we detail on the implementation of CLASH. It is the underlying system that provides a full cut from providing users a SQL-style query front-end to optimizing the query and producing a Storm topology with runtime components that can actually execute the given query or queries.

Finally, Chapter 9 concludes the thesis and provides an outlook.

Chapter 2

Background and Preliminaries

This chapter presents background information which can be useful for understanding later aspects of the thesis. We discuss continuous queries in general and focus on window semantics as well as time. After that, we look at join computation in distributed environments, which have been extensively studied for bulk operations. We provide a high-level introduction to Apache Storm as our approach of the later chapters is implemented on top of it. Finally, we give a brief introduction to the join order optimization problem.

2.1 Continuous Queries

A stream is a never-ending inflow of tuples. Thus, query semantics like used in relational database systems are not straightforward to apply. Most prominently, this is due to the strong presence of time. Consider a simple aggregation query that counts all tuples; the result of this query changes every time a new tuple is seen. To cope with that problem, **continuous queries** were proposed.

A continuous query is formulated either in a continuous query language like CQL or similar SQL derivatives like KSQL [11, 39], or in a programming language like Java or Python (e.g., in Storm, Flink, or Spark [1, 8, 10]). A stream processing system is responsible for executing the query. In contrast to static database systems where queries are instantaneous events, continuous queries have a lifespan: They are registered, continuously produce results, and eventually they are deregistered again.

Arasu et al. define a stream S as "a bag of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the timestamp of the element." [11] The timestamps define an order on the tuples which can be used to interpret ranges between two timestamps as static relations. Such ranges are commonly used to define **windows** of tuples. A window is useful for batching communication with the user's client application, but can also be a necessity, e.g., to provide a context for aggregation queries.

2.1.1 Windows

Consider the example in Figure 2.1 where tuples s_i are observed at the timestamps indicated by the timeline. The first windowing strategy shown is called **tumbling window**: periodically (in the example, every 10 time units) a new window is created which contains a part of the previously seen tuples (in the example, all tuples from

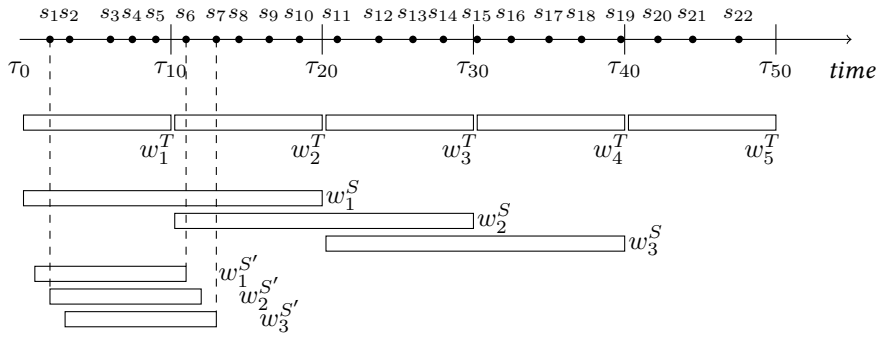


Figure 2.1: Tumbling and sliding windows on a stream.

the last 10 time units), w_1^T contains tuples s_1 to s_5, w_2^T contains s_6 to s_{10} and so on. Such tumbling windows are partitions of time. When the period P (also called range) of the windows is smaller than the length, we get a **sliding window**. In the example, the window period is 10 time units, the window length 20 time units, thus tuples s_{11} to s_{14} belong to both windows, w_2^S and w_3^S. The content of two sliding windows at times τ and τ' only changes, if a new tuple arrives between τ and τ', or a tuple has a timestamp between $\tau - P$ and $\tau' - P$. In the figure, window $w_1^{S'}$ as of τ_{11} contains tuples s_1 to s_6. Window $w_2^{S'}$ as of τ_{12} contains tuples s_2 to s_6, so compared to the previous window, s_1 was removed. Then, window $w_3^{S'}$ as of τ_{13} contains the fresh observed tuple s_7. These smallest distinguishable windows are the result when windows slide continuously instead of incrementally by P and in this thesis, we focus on such continuously sliding windows.

The aforementioned windows are **time-based**, as they compute their contents based on the current timestamp and the tuples' associated timestamps. It can happen, that such windows are completely empty (if no tuples arrive in the time boundaries) or that they contain a massive amount of data, too much for downstream operators to handle. Another way is, to use **count-based** windows. They contain a set number of tuples, and can again be overlapping or not. The time between two count-based windows depends on the input behavior of incoming tuples: if tuples are arriving at a higher rate, more windows are produced, and if no more tuple arrives, the last window might never end.

These, time-based and count-based, are the primary window variants supported by most systems. Further, these variants can be extended, for example, Flink allows adding timeouts to count-based windows, or offers **session windows**, where tuples belong to the same window if the gap between them is not too large [8].

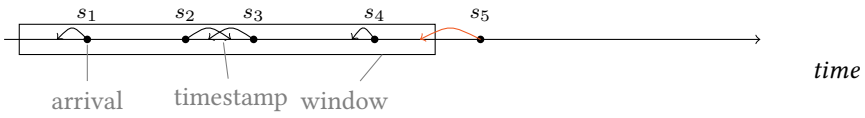


Figure 2.2: A tuple which logically belongs to a window arrives late.

2.1.2 Time and Delays

So far, we assumed that tuples arrive sequentially and contain monotonically increasing timestamps. In reality, this might not necessarily be the case. The timestamp could be assigned by a different system, e.g., a sensor network, where some sensors have connection issues and thus their data arrives late at the processing system. Another case might be that data is gathered in parallel on different sources and tuples arrive out-of-order. Here, the delay might be smaller, but can still affect the result. A timestamp that arrives at the stream processing system as part of the data tuples is also called **event time**—in contrast to timestamps assigned by the stream processor (**system time**). The stream processing system might run on multiple physical machines and due to divergence of the local clocks it is not trivial to interpret timestamps.

Consider a window at time τ. It contains tuples with timestamps $\leq \tau$. If this window would be exposed to downstream operators immediately, and another tuple with timestamp $\tau' < \tau$ arrives after the window is output, the query result would be erroneous. The previous window could be updated, the window could be output a second time, the tuple could be dropped, or the tuple could be added to the next window. This problem is illustrated in Figure 2.2. The arrival time of the tuples is displayed by the dots, and the timestamp of each tuple by the arrow head. Tuple s_1 arrives at a time later than the timestamp, s_2 and s_3 are even swapped. This might be interesting for some queries, but if the logic is only evaluated at the end of the window, this is no problem. However, s_5 arrives after the window is finished (hence, evaluated and results reported), but logically the tuple belongs to the window. Such cases need to be handled by the systems. Flink, for example, introduces an allowed lateness [8], and we introduce a buffer for delayed tuples in Section 5.4.

2.2 Distributed Join Computation

A (binary) join operation takes two sets of inputs, R and S, and produces an output from the Cartesian product of the inputs, $R \bowtie S \subseteq R \times S$. Each element from the result has the form $r \circ s, r \in R, s \in S$. In order to produce the result, r and s need to "meet" each other. In very basic nested-loops joins, this is ensured as each loop iterates over the entire relation, thus it can enumerate the entire Cartesian product. An algorithm that does not enumerate the entire Cartesian cross product might still

be able to produce the correct join result, if it understands the join predicate at hand well enough. For example, a hash join only compares tuples where the hash value of the join attributes is the same.

For *distributed* join computation, where tuples are placed on different nodes in a network, we conclude that a join strategy must make sure, that joinable tuples are placed in such a way in the network, that all tuples r and s are at the same node if they belong to the join result.

2.2.1 Parallel Equi-Join Computation

We first look at a prominent special case that most of the time does *not* need to enumerate the entire cross product, the **equi join**. A very basic method of computing an equi join is the parallel version of the grace hash join as described by DeWitt and Gerber [22]. Tuples of both relations are horizontally partitioned on a set of compute nodes. The relations are hashed into buckets and these hash buckets are sent to remote nodes such that all buckets with the same hash value end up at the same node.

Figure 2.3 illustrates this approach for joining two relations R and S. On the left, there are two storage nodes where tuples of the both relations are stored, node $storage_1$ stores partitions R_1 and S_1, the other node stores partitions R_2 and S_2. If the hash value for a tuple is i, this tuple is sent to $worker_i$ for joining. The workers themselves are responsible for locally joining the arriving tuples and produce the result.

This is the basic recipe for equi-join computation in other contexts. For example, in MapReduce [20], the two relations are handled by the map function that outputs pairs of `(join-key, (relation-tag, tuple))` and the reducers compute the cross product for each join-key (also reduce-side join or repartition join [16]). This strategy is also used in Spark's shuffled hash join, where first, a partitioned dataset is created and then in these partitions the join is computed.

While there are differences in the systems' underlying data models and APIs available for implementation, the common part is, that for a tuple of the input relation, we need to know the address where it will meet with all potential join partners.

In the parallel hash join, every tuple was sent to exactly one worker node. This was possible, because the equality predicate allows drawing very specific conclusions about potential join partners when examining a single tuple. In particular, for a query $(R, S, R.a = S.a)$ we know that the join partners of a tuple $r = \{a : 5, \dots\}$ are of form $s = \{a : 5, \dots\}$.

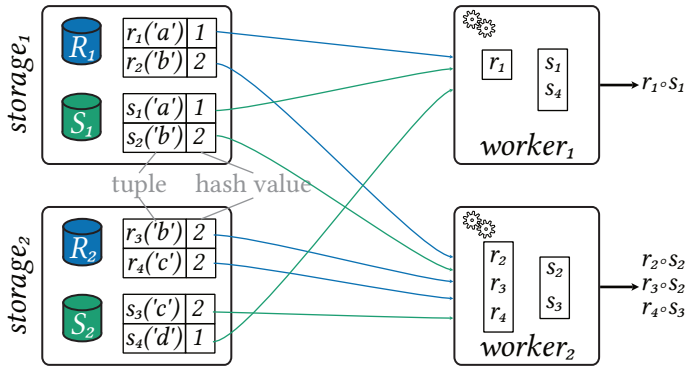


Figure 2.3: Illustration of a parallel hash join with two involved relations.

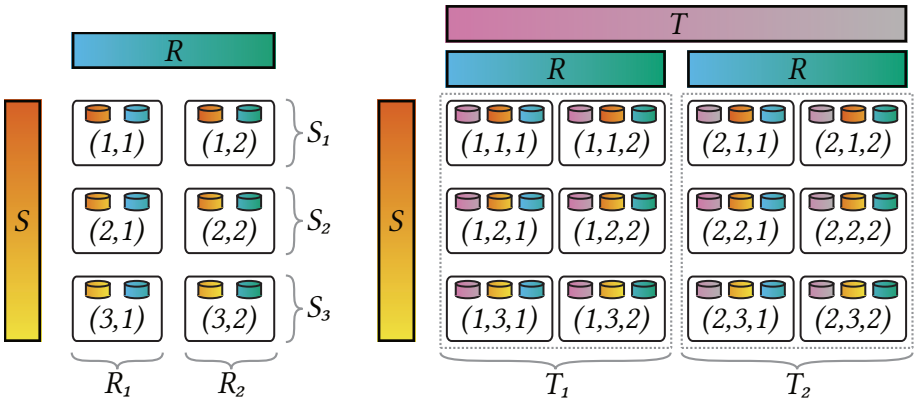
2.2.2 Matrix and Hypercube for Theta Joins

Properties of other predicates can also be exploited, for example in a band join. Here, both tuples lie within a band of certain width, i.e., the difference Δ between the join attributes is bound. For a query $(R, S, |R.b - S.b| \leq 2)$ we know that join partners of tuple $r = \{b : 5, \dots\}$ are of form $s = \{b : \mathbf{b}\}$ with $\mathbf{b} \in [3, 7]$. When not hashing the tuples to worker nodes, but assigning value ranges, this knowledge can be exploited to send tuples to a small set of worker nodes where join partners can be found.

If the predicate θ is arbitrary, for example because it is expressed as user-defined function, we cannot infer properties about the join partner. Hence, a tuple from one relation must see all tuples of the other relation in order to guarantee a correct result. One prominent approach to do this, is to arrange the worker nodes in a **matrix** [52], and send tuples of one relation to all workers in some column, and the tuples of the other relation to all workers of some row. This way, each tuple gets replicated several times and the correct join result can be produced.

Figure 2.4 shows two examples for matrix-based partitioning schemes. The boxes represent workers and are annotated with indices of the partitions that are assigned to them. In Figure 2.4a, the join between relations R and S is computed on 6 worker nodes. R is partitioned into two partitions $R = R_1 \cup R_2$ and R_1 is sent to the left column of workers, R_2 to the right column of workers. S is partitioned into three partitions $S = S_1 \cup S_2 \cup S_3$ and the first partition is sent to the first row, the second partition to the middle row, and the third partition to the lower row. If we now inspect a tuple $s \in S$, it is in exactly one of the partitions of S, and thus it is sent to all workers in one row. The workers in this row receive each partition of R, and thus entire R. Analogously $r \in R$ sees all tuples of S and thus the correct join result can be computed.

A join using multiple relations can be conducted using a generalization of this matrix assignment, a **hypercube** [15] where each dimension represents the partitions



(a) Matrix (b) Hyper cube

Figure 2.4: Matrix assignment for binary join and hypercube assignment for ternary join.

of a relation. Figure 2.4b extends the previous example to three relations, R, S, and T. Now there are twelve workers, the partitions of R and S are the same, and T is partitioned into T_1 and T_2. T_1 is assigned to the left block of workers, T_2 to the right one. Again, a tuple $s \in S$ is assigned one row. In this row, it meets all four possible combinations of the partition indices of R and T. Tuples of R can go on even or odd rows, and tuples of T to the left or the right block. Still, each tuple of one relation sees all partitions of the other relations, thus also this hypercube scheme is correct.

2.2.3 Stream Joins in General

In traditional database systems, a theta join between two relations R and S is computed using the nested-loops join: Each pair of $r \in R$ and $s \in S$ is generated and $r \circ s$ (i.e., the concatenation of r and s) is added to the result set iff a given predicate $\theta(r, s)$ is satisfied. In this case, the relations are assumed static for the time of query evaluation. This means, once a tuple $r \in R$ has been probed completely against relation S, it does not have to be considered again. For data streams, however, this is not true, as the relations are continuously growing: If a tuple $r \in R$ is probed against tuples of another streamed relation S, newly arriving tuples of S cause r to be reconsidered, because only a **prefix of the relation** (i.e., tuples of this relation seen so far) is known. Simply put, a key difficulty is to ensure that all potential join partners eventually meet—once and only once.

In general, for relations $R_1, \ldots, R_n$, the join predicate θ has to hold for all possible n-tuples. Otherwise, the tuple can be discarded as it is not part of the join result. As mentioned earlier, we assume that θ is the conjunction over binary predicates $\theta_{i,j}$ which have to hold on two relations R_i and R_j.

To illustrate the general problem behind processing joins of streamed relations, let us walk through an example. Consider a distributed system of compute nodes, each containing a prefix of previously seen tuples of some part of the incoming relations. Now, let us look at one specific tuple that arrives in one of the relations. When a new tuple arrives, some of the buffered tuples are potential join partners. Thus, they have to meet at some node in which the predicate of the query is evaluated. Let us say the tuple that just arrives is $r \in R$ and the query at hand involves two more relations, S and T. There are nodes that contain the prefix of S, respectively, T. To determine all result tuples that contain r, it is sufficient to first send r to the nodes storing S, where the partial join $R \bowtie S$ is computed. In case r does not find any join partners, the computation can be stopped and nodes that store tuples of T do not have to be contacted at all. Further, there could be nodes buffering the intermediate results from the partial join (with respect to the desired three-way join) $S \bowtie T$. Then, it is sufficient to send r to these nodes in order to find join partners.

2.3 Apache Storm

Apache Storm [1, 56] is a popular framework that allows building and deploying application logic in form of operator topologies to machines in a compute cluster. The available tuple routing primitives between individual nodes allow expressing arbitrary query plans, while the runtime layer of Storm handles execution and comes with desired properties like fault tolerance.

2.3.1 Components and Configuration

Storm consists of the following main components as programming abstractions for developing distributed real-time applications:

- **Spouts**, source components where data enters the topology. They implement a method which is called repeatedly and can emit tuples.
- **Bolts**, operational components where data tuples arrive and are transformed and sent along. They implement a method which is called when a tuple is received and can emit tuples.
- **Streams**, connections between spouts and bolts. There are different grouping types of streams, and depending on the type, the tuples are sent to different instances of bolts.
- **Topology**, a configuration of spouts, bolts, and streams, which can be deployed to a Storm cluster.

Figure 2.5 shows a condensed example of how these components are defined to form a program that counts how often a name was greeted with "Hi". First, the spout is defined. In its `open` method it connects to some http stream, where it receives all messages. When the `nextTuple` method is called, the next http item is transformed into a pair consisting of the salutation and the name. These two elements are then output using Storm's emit method. The FilterBolt reads arriving tuples and only if the first entry is "Hi", an output tuple with the name is emitted. Here we also see in the `declareOutputFields` method, that output tuples have field names; in this case the only entry of output tuples is a field called `"name"`. The declaration of output field names is mandatory, but we skip this for other classes and other aspects for conciseness. The CountBolt has a state consisting of the map where the counts of the observed names are stored. When a tuple is received the count of the contained name is increased and initialized if it was not observed before. Finally, the main method shows how a topology is built: spout and bolt instances are registered with a name and optionally a degree of parallelism, e.g., the FilterBolt is registered as `"filter"` with a degree of parallelism of 3. Using the name of the component a stream can be registered. In this case, a stream with shuffle grouping from Spout to FilterBolt and a stream with fields grouping from FilterBolt to the CountBolt.

2.3.2 Stream Groupings and Parallelism

Spouts and bolts are instantiated when a topology is built, then serialized and copied to the compute nodes of the Storm cluster where they execute. In the previous example, the FilterBolt is copied three times and the CountBolt five times. Such a copy is called a **task**.

When defining a topology, we establish streams between spouts and bolts, but the stream grouping defines the concrete tasks that receive tuples. The most important stream groupings are

- **shuffle grouping**, where tuples are sent to a task in round-robin fashion,
- **all grouping**, where tuples are sent to all tasks,
- **fields grouping**, where tuples are sent to the same task, if their grouping attribute has the same value.

While shuffle and fields grouping send a single tuple independently of the degree of parallelism of the receiving bolt, all grouping emits as many tuples as there are tasks of the receiver.

Figure 2.6 shows the stream grouping in the previous example in effect. The rounded rectangles represent the spout and bolts, the smaller squares represent tasks. For the spout there is one task by default. For filter and counter there are three and

```
class Spout : BaseRichSpout() {
    lateinit var httpStream;
    override fun open() {
        // connect to http stream
    }


    override fun nextTuple() {
        val (greeting, name) = transform(httpStream.next())
        emit(Values(greeting, name))
    }

    // produces pairs like ("Hi", "Tom"), or ("Bye", "Mary")
    fun transform(String): Pair<String, String>
}

class FilterBolt : BaseRichBolt() {
    override fun execute(input: Tuple) {
        if (input.getString(0).contains("Hi")) {
            emit(Values(input.getString(1)))
        }
    }

    override fun declareOutputFields(declarer: OutputFieldsDeclarer) {
        declarer.declare(Fields("name"));
    }
}

class CountBolt : BaseRichBolt() {
    val count = mutableMapOf<String, Int>()

    override fun execute(input: Tuple) {
        val name = input.getString(0)
        count[name] = count.getOrElse(name, { 0 }) + 1
    }
}

fun main {
    val builder = TopologyBuilder().apply {
        setSpout("spout", Spout())
        setBolt("filter", FilterBolt(), 3)
            .shuffleGrouping("spout")
        setBolt("counter", CountBolt(), 5)
            .fieldsGrouping("name")
    }

    // submit topology to cluster
}
```

Figure 2.5: Example Storm topology that counts which name was greeted how often.

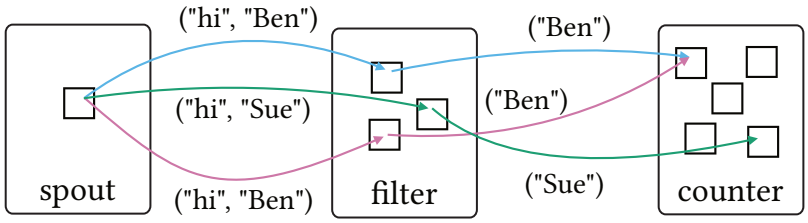


Figure 2.6: View of the parallelism of the topology of Figure 2.5.

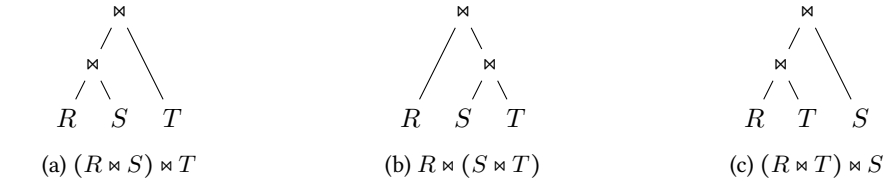
five tasks, as defined in the topology. The spout and filter are connected using shuffle grouping, thus, the tuples end up at different tasks, especially both ("hi", "Ben") tuples. It would be equally valid for all tuples to land at the same task of the filter bolt. The filter and counter are connected using fields grouping, where the first and only field of the tuple forces that both ("Ben") tuples end up at the same task. It would be equally valid for the ("Sue") tuple to arrive at the same task as the other two, but the ("Ben") tuples can not land on different tasks.

2.4 Join Order Optimization

A (binary) join operation has two relations as input and outputs a new relation containing combinations of the input relations' tuples. The result size of a join can be anywhere between no tuples and all combinations, i.e., the product of the sizes of the input relations. This is an important difference to other common operations, like filtering or aggregation, where the result size is at most as large as the input.

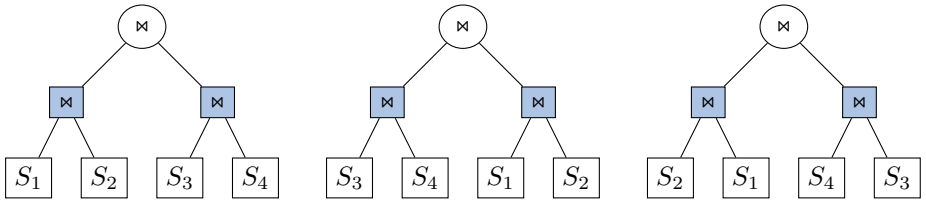
Multiple join operations executed in sequence are associative and thus the order in which they are executed can be arbitrarily chosen. Consider a join query over three relations, (R, S, T, θ) with some conjunction of binary predicates θ. Due to associativity we can compute the join as $(R \bowtie S) \bowtie T$ and $R \bowtie (S \bowtie T)$. Using schema information, a join operation is also commutative, thus also $(R \bowtie T) \bowtie S$ is a feasible order of the joins.

Different options are commonly displayed as trees where the leaves represent input relations, inner nodes represent intermediate or partial join results, and the root signifies the query result. The trees for the three aforementioned join orders are shown in Figure 2.7.



(a) $(R \bowtie S) \bowtie T$ (b) $R \bowtie (S \bowtie T)$ (c) $(R \bowtie T) \bowtie S$

Figure 2.7: Different possibilities of constructing query plans for three input relations.



(a) $(S_1 \bowtie S_2) \bowtie (S_3 \bowtie S_4)$ (b) $(S_3 \bowtie S_4) \bowtie (S_1 \bowtie S_2)$ (c) $(S_2 \bowtie S_1) \bowtie (S_4 \bowtie S_3)$

Figure 2.8: Three different permutations of the base relations under the same tree structure.

2.4.1 Problem Complexity and Subclasses

Given a list of $n + 1$ relations, for computing the overall join in a sequence of binary join operations the number of orders is the n^{th} Catalan number, C_n [55]. For a list with $n + 1 = 3$ elements like $[R, S, T]$, there are $C_2 = 2$ such sequences, and these are shown in Figure 2.7a and 2.7b. If the list is sorted differently and then placed under the same tree, a solution like 2.7c can be crafted.

In principle, there are $(n + 1)!$ different ways of sorting such a list; however, this is only relevant, if the join operation is not commutative. For example, if nested-loops joins should be executed for $R \bowtie S$, then R is the outer relation and S is the inner relation, not the other way round [27]. For this thesis, the join operation is commutative, so, $R \bowtie S$ and $S \bowtie R$ are equivalent. Consider relations S_1 to S_4 with join trees shown in Figure 2.8. The shape of the three example trees is the same, but due to commutativity, one tree can be transformed into the other: Tree 2.8a can be transformed into Tree 2.8b by changing the inputs of the root operator.

Further, in this thesis we are researching multi-way trees. Thus, there are in fact more valid tree shapes available for answering a certain query, e.g., the one in Figure 2.8c, where the root node has three inputs. To the best of our knowledge the exact number of different multi-way trees modulo commutativity is not known. However, C_n serves as non-strict lower bound which already shows that it is prohibitively expensive to completely explore this space of trees.

Instead, optimizers can narrow down the search space by only examining a certain shape of trees (e.g., left-deep trees) which may yield a sub-optimal solution, or exploit

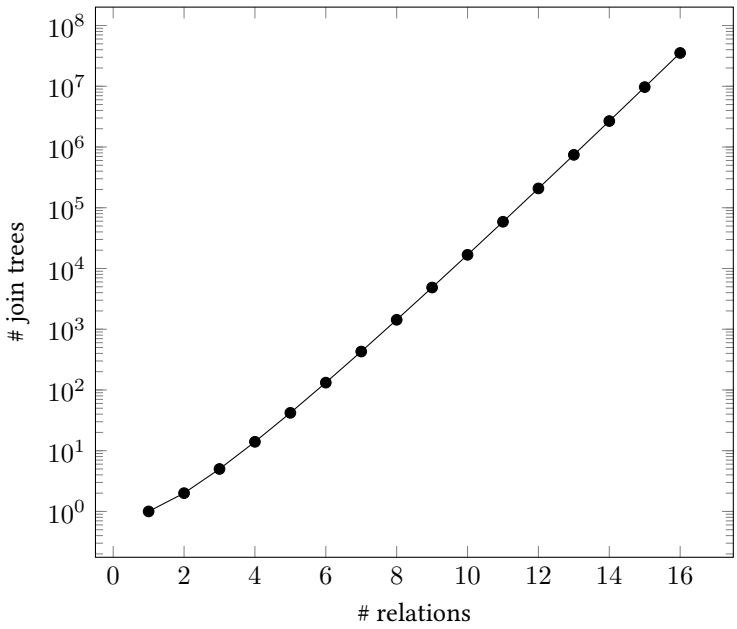


Figure 2.9: Growing solution space for growing query size.

features of the query (e.g., star-queries can only be executed using left-deep trees).

Consider the graph shown in Figure 2.9 with the number of relations involved in a query on the x-axis and the number of potential join trees for answering such a query on the logarithmic y-axis. The entire problem space represented by the line of C_n grows exponentially.

Chapter 3

Related Work

This thesis is related to several lines of research which are partially overlapping. First, in Section 3.1, we discuss literature on general stream processing. In Section 3.2, we look into prior work on stream processing, including joins, which was done without distribution in mind. Distributed join computation in a bulk synchronous processing context like MapReduce was studied with different aspects highlighted such as allowing arbitrary predicates for binary joins, optimizing the number of rounds for multi-way equi-joins, or dealing with skewed loads for individual workers which we discuss in Section 3.3. Following that, we discuss in Section 3.4 works which target both, distributed join computation and stream processing, and in Section 3.5 multi-way join computation that was explored in non-streaming contexts.

The term "streaming" sometimes refers to the fact that datasets are read in a certain sequence and the stream of tuples is fed into an operator pipeline. In case of distributed components also that data is read in a streaming fashion from a remote data source. The data set that is streamed already exists completely at some place and the fact that it is accessed in a streaming fashion is a purely technical aspect.

In this thesis, however, we consider streaming data to be *new* data that was only recently appended to some dataset. For example, a temperature sensor generates a new data point every second, and such a sequence of data points is what we call a stream.

3.1 Stream Processing in General

For streaming data in general, Viglas and Naughton [59] propose the use of rate-based optimization rather than classical cost-based optimization. The reason for this step lies in the continuous nature of data streams, where relation sizes are infinitely large. With the switch towards rates they also present two new optimization goals: (1) select the plan that can produce the most tuples upto a given point in time, and (2) select the plan that produces a given number of results the fastest. We employ rate-based optimization, too, and extend that view to intra-operator components.

Ayad and Naughton [12] use rate-based optimization techniques and differentiate between feasible and infeasible queries and plans: A feasible plan can handle all tuples while an infeasible one cannot. Still, an infeasible plan can be executed and an approximate solution presented to the user. In this case, load-shedding strategies need to be applied [32, 19]. Such strategies could also be integrated with our approaches,

however, we will not examine load shedding in this work.

In [49], Meehan et al. discuss ETL workloads in the context of streaming systems with the goal of reducing latency; and in [50], Meehand et al. integrate these workloads with ACID-style transaction processing. For XML-processing, Hong et al. [36] research joins between documents and between nodes of the same document with multiple long standing queries. Karimov et al. [42] present AStream, a system for sharing resources for multiple streaming queries. They share parts of the history of joins however compared to our approach, only if exactly the same joins are used in different queries and they ignore partitioning.

3.2 Local Stream Join Processing

Golab and Özsu [30] process n-ary stream joins on a single machine using n nested-loops joins, where the join order is determined using the arrival rate of the streams and selectivity of the predicates. They present two major evaluation strategies, **eager** evaluation and **lazy** re-evaluation. For eager evaluation, they determine for each tuple arriving in a sliding window a join order and then compute the join result for that tuple. For lazy re-evaluation, they only compute joins after a set time period τ, either in n nested-loops joins or as generalized plan in a single nested-loops join. They also use hash-based implementations, compared them to their pure nested-loops lookup-based, and found the hash-based implementation to be more efficient. The lazy evaluation approach is similar to the micro batching concept which is used later in Spark Streaming [10].

Hammad et al. [33] present two algorithms for processing windowed multi-way joins, a response-time optimized and a throughput optimized algorithm. The former computes a join result as soon as a new tuple arrives by recursively joining tuples to the next relation of a global join order. The latter operates on a working set of tuples by first determining window boundaries and then computing joins, thus saving on window evaluation. While this second approach offers higher throughput, it increases latency. The two algorithms are described in a centralized setting and there is no consideration of how such algorithms could potentially be executed in a distributed fashion. The algorithms are, however, oblivious to the matching predicate, and, thus, not bound to simple equi joins.

3.3 Distributed Join Processing

Distributed join computation was extensively studied in the context of many-core systems. Albutiu et al. [6] worked on a parallel join algorithm with the goal of scaling linearly with the number of cores. Barthels et al. [14] analyzed performance on

joins of datasets on a high-performance computing infrastructure with 4,096 cores and focus on sort-merge and hash joins. Such approaches can be described by the bulk-synchronous parallel (BSP) computing model [57] where the algorithms are evaluated on p components in parallel in a sequence of super steps and with defined communication between the super steps. BSP programs are typically used for computations on complete datasets, while of our approach operates on individual tuples of continuously arriving datasets.

A popular instance of BSP is MapReduce [20], where a map function $\mathcal{D} \rightarrow K \times V$ is applied to all data items in $\mathcal{D}$ (which resides on multiple machines) in parallel. The output is grouped onto the available machines according to the value of K, where a reduce function $K, V^+ \rightarrow \mathcal{D}'$ computes a result. For MapReduce, Afrati and Ullman explain in [5] how to optimize multi-way equi-joins by minimizing the communication cost. In [4], Afrati et al. propose a multi round algorithm for further reducing communication cost. Zhang et al. [66] also decompose multi-way joins into a sequence of map reduce join operations, however, they explicitly include theta joins. Focusing on binary theta joins in MapReduce, Okcan and Riedewald [52] arrange reducers in a matrix and the mappers' job is to choose a column or row of the matrix in a way to guarantee correctness of the result and ideally also efficiency of the join computation.

Lang et al. [44] discuss diverse optimizations for cloud-based computation of joins where the system is configurable to use more or less cost or a given budget. This is similar to our approach of setting hard limits, and also they further discuss lowering quality of the query result in exchange for better cost.

3.4 Distributed Stream Join Processing

This join matrix approach can also be used for computing stream joins [28], where a relation R gets replicated across n machines, and each tuple of relation S is forwarded to one of the n machines, and vice versa. Generalizing this scheme to multiple relations resembles a hyper cube where relations get replicated to machines for multiple dimensions [15]. A system that implements join matrix and hypercube is Squall by Vitorovic et al. [60]. Squall leverages different variations of the join-matrix model where each matrix cell produces a fraction of the join result. By choosing the matrix cell sizes such that their perimeter is minimal, Squall achieves low latency while getting better resource utilization. For matrix-based approaches, there is further work on adjusting dimension splits for handling skew [29, 64].

Lin et al. [47] on the other hand, propose a tuple-routing scheme that avoids replication of data stream tuples. This BiStream approach is explained in more detail below.

Zhou et al. [67] propose PMJoin, an approach for minimizing the communication cost between computing nodes when evaluating a multi-way join query with a focus

on equi joins. They introduce a heuristic-based algorithm for deciding the join order and explicit placement of operators in a computer network. This algorithm computes for n distributed data stream sources and a given query plan in $O(n^2)$ a linear join tree.

Oguz et al. [51] propose changing the implementation during query answering from symmetric hash join to bind join and back, depending on arrival rates and result size. Madsen et al. [48] describe the scaling of operators in distributed streaming engines. For extremely skewed input, Rödiger et al. [53] and Li et al. [46] split the handling of heavy hitters, i.e. single tuples with a very high selectivity, from the rest of the tuples.

Yang et al. [65] propose cost-based optimization where operators are shared between plans for multiple queries if they produce the same or implied output streams. Also for multiple queries, Jonathan et al. [41] show multi-query optimization where multiple data centers are involved and slower inter-dc-communication is respected. They introduce operator sharing strategies for saving both, computation and communication.

3.4.1 BiStream

The BiStream [47] operator can be interpreted as a special case of the MultiStream operator which we present in the later chapters of this thesis, since they work identically on a binary join.

For computing a join between streamed relations R and S, for both relations, distributed tasks $R_1, \ldots, R_m =: \mathcal{R}$ and $S_1, \ldots, S_n =: \mathcal{S}$ are registered. Arriving tuples of R (S) are randomly sent to one of these tasks R_i (S_i) where they are stored for later arriving probe tuples. These probe tuples are sent from one arriving relation to the other relation's tasks, so tuples from R are sent to all tasks S_i and vice versa. To be more specific, a tuple $r \in R$ arriving at time τ is sent to all tasks $\mathcal{S}$ where the result $r \bowtie S$ with all tuples from S that arrived before t is computed. The same tuple r is sent to one of the randomly chosen tasks of $\mathcal{R}$ where it is stored and thus $r \bowtie S$ with all tuples of S that arrive later than τ can be computed.

BiStream's correctness is guaranteed using timestamps. Stored tuples are only joined with probe tuples that have a higher timestamp. This makes it necessary to cope with delayed tuples: these are store tuples that arrive after a probe tuple with a higher timestamp was already processed. Probe tuples are kept at each task for some period of time to allow them to be probed against late arriving store tuples.

The BiStream approach was presented for two inputs and it was hinted that multiple BiStream operators can be combined to operator trees, but details were not given. For the experimental evaluation, queries with multiple inputs were preprocessed such that only a single binary join over materialized intermediate results can compute the final result.

3.5 Multi-way Join Processing

The use of multi-way joins in conventional database systems is discussed by Henderson and Lawrence [34]. They compare different multi-way join algorithms for equi-joins and conduct experiments using PostgreSQL, with the conclusion that multi-way join operators can be in particular useful for star queries.

Joglekar and Ré [40] propose using information on the multiplicity of values to optimize multi-way joins, also limited to equi-joins, and not considering distributed computation (although some results are of generic nature).

Gomes et al. [31] propose changing roles of relations in a binary join tree. Similarly, in DBMSs adaptive processing techniques are employed, e.g., for long running queries where the initially selected plan turns out to be suboptimal. Li et al. [45] are also changing the roles of relations in the join tree. Kolchinsky and Schuster [43] apply join optimization techniques for complex event processing systems where many patterns are registered simultaneously.

Chapter 4

The Architecture of CLASH

In this chapter, we describe the architecture of CLASH and build the basis for understanding the join approaches of the subsequent chapters. We explain the high-level concepts and the relation to Storm as underlying stream processor. In Chapter 8, we will further detail on the implementation as well as discuss applicability to other stream processing systems.

CLASH's architecture is tailored to join computation between data streams in a local compute cluster and intentionally ignores other aspects like connecting to persistent databases, high-availability, or wide-area operator location—all tackled by other research [21, 37, 41].

CLASH spans a topology which is a directed multi graph of stores as well as source and sink nodes interconnected by edges. Input arrives tuple by tuple at some source and is sent along edges to stores. Upon receiving a tuple, a store can add this tuple to its local storage, produce a new tuple to send it to another store, or do both. When a query is registered, CLASH's optimizer is responsible for creating the topology and configuring all nodes.

Example

Figure 4.1 depicts an exemplary topology for computing the join between three relations, R, S, and T. It contains four stores, the R-, S-, and T-store, as well as the RS-store. We name the stores according to the relations they contain. The first three stores represent exactly the input relations while the RS-store contains tuples of the intermediate relation $(R, S, \theta_{R,S})$. Input and output are secondary to the actual representation, we only rely on a tuplewise interface to read tuples and emit join results as soon as possible.

The join result is produced as follows. All arriving tuples of R are sent to the R-store where they are locally stored. A copy of each R-tuple is sent to the S-store where it probes the previously arrived S-tuples to find join partners. All intermediate result tuples are sent to the RS-store for storing and to the T-store again for probing, this time to compute final results. All final results are then sent to the output which represents downstream consumers of the join result. This works analogously for tuples of S, tuples of T can take a shortcut and leverage the RS-store: input tuples from T are sent to the RS-store for probing and can produce final results in a single hop.

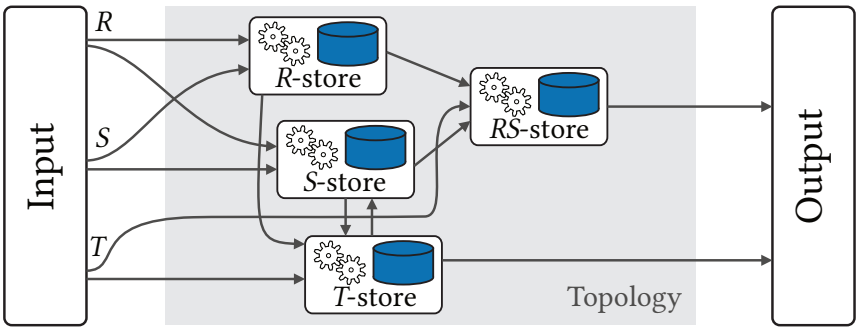


Figure 4.1: Data enters the topology over the input on the left and is spread over the stores where tuples are processed and sent to other stores and eventually to the output on the right.

This example highlights two important aspects of a CLASH topology:

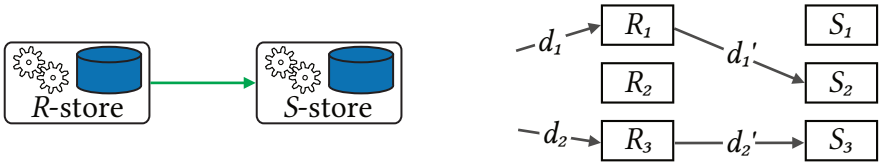
1. there **may be multiple edges** between two nodes and
2. a topology **may contain cycles**.

Consider the R-store which has two incoming edges from the input. This is necessary to differentiate between tuples of relation R which should be placed in the local storing data structures, and tuples of relation S which should be probed against the stored tuples and potentially produce a join result. Typically, topologies are described as acyclic [1] graphs, or cycles are reserved for fix point iterations [8]. In our case the cycles are only visible in the static view on a topology. However, we construct the topology such that the *trace* (i.e., the sequence of stores that are visited as reaction to a tuple arriving at the topology) of a tuple does not contain a cycle. Consider the cycle between the S- and the T-store in Figure 4.1; either a trace contains the edge from S-store to T-store, or from T-store to S-store, or none of them.

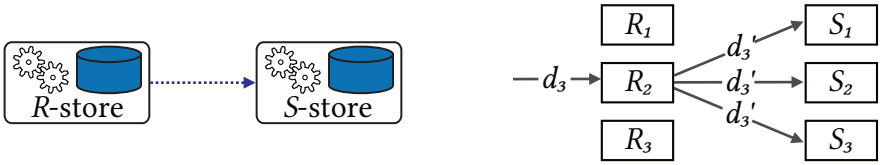
4.1 Stores and Edges

A store contains a relation that can be interpreted as the result of a query. For the example above, these are the three identical queries over R, S, and T, and the join query $(R, S, \theta_{R,S})$. If we inspect the contents of a store at some point in time τ, we can expect it to consist of all tuples that belong to that relation up to τ. This way, we can use the RS-store for probing with an arriving tuple of T.

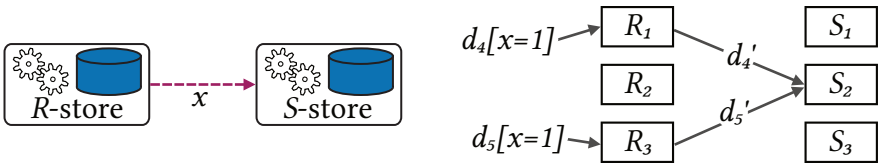
A store is a distributed component, implemented as stateful bolt in terms of Storm. This means, multiple so-called tasks can run instances of the same store on different machines in a compute cluster. The state, containing the tuples of the store's relation,



(a) A shuffle edge, the target is chosen randomly.



(b) An all edge, all S-tasks are targets.



(c) A group-by edge, the target is chosen based on the value of x.

Figure 4.2: Illustration of communication patterns of the three edge types if installed between R- and S-store. Topology-view on the left, task-level deployment if both stores are distributed on three tasks on the right.

is partitioned over these tasks, i.e., each tuple of a relation resides on exactly one task. This partitioning can be done randomly or based on some attribute value of the stored tuples.

An edge connects two stores and indicates that tuples are sent along this edge. The type of edge used differs depending on the operation and the partitioning strategy of the receiving store. CLASH edges correspond to named streams in Storm.

In Figure 4.2 we see three possibilities for edges, where the left part shows how we depict such edges in a topology with stores for relations R and S, and the right part shows a trace of tuples with both stores distributed on three tasks each. In the example, data items d_i arrive at the R-store and as a result, derived data items d_i' are sent to the S-store. The shuffle edge in Figure 4.2a is depicted with a green solid arrow. After d_1 is processed by R_1, d_1' is sent to S_2, and after d_2 is processed by R_3, d_2' is sent to S_3. However, the targets could be different, e.g., they could be swapped, or both data items could have ended up at the same task. The all edge in Figure 4.2b is depicted with a blue dotted arrow. When d_3 arrives at R_2 it generates as result data item d_3' and a copy of this item is sent to every S-task. The group-by edge in

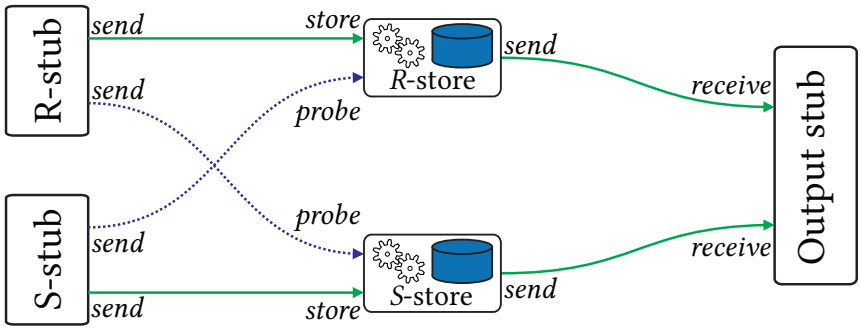


Figure 4.3: A simplified physical graph for computing the join between two relations, R and S.

Figure 4.2c is depicted with a purple dashed arrow and an indication of the attribute used for grouping. We see d_4 and d_5 arriving at different R-tasks, but with the same x-value. In this example, we assume that in d'_i the x value is copied from d_i, thus d'_4 and d'_5 have the same x value and hence, they are both sent to S_2.

4.2 The Physical Graph

Creating such a topology by hand is error-prone and thus CLASH relieves users from this task. The input is a declarative query as well as a configuration for the optimizer. The optimizer is then responsible for creating a physical graph. This graph is a representation of the stores, edges, and their configuration which is used as template for creating a Storm topology. Since it is more abstract than latter, it can also be used to deploy the computation to other platforms like Kafka or create a custom Akka application for answering the query.

Figure 4.3 shows a simple example for a physical graph for computing a join between two relations R and S. For both relations, an input stub is created and connected to both stores. Input from a relation travels along a shuffle-edge to its relation's store, and along an all-edge to the other store. Join results are sent along a shuffle-edge to the output stub. For each outgoing and each incoming edge, a rule indicates which actions are done when sending or receiving a tuple. In the figure, the input stubs just send all incoming tuples over the edges to the stores. At each store, the type of an incoming rule indicates whether a tuple should be stored in the local prefix, or probed against the prefix and potentially produce a join result. Due to the outgoing shuffle rule of the stores, join results are forwarded to the output stub where a receive rule handles incoming tuples.

When CLASH translates a physical graph into a Storm topology, input stubs are

replaced with spouts, and the output stub with another bolt. However, this optimization result is also generic enough that it can easily be plugged into a bigger topology where stubs are replaced by other components.

Chapter 5

Streaming Full History Joins

We start explaining our approach to join computation by examining joins over the full history of seen tuples where an arriving tuple $r \in R$ can join with any previously seen tuple $s \in S$. In reality, such an approach quickly becomes infeasible, because an ever-growing amount of tuples would have to be stored. This is commonly avoided by imposing a temporal restriction to the joinable tuples, such that only the recent portion of tuples needs to be accessed. However, in experimental research, looking at full-history computations frees from the need of discussing tuple eviction strategies and enables focussing on the online computation of joins. A full history join can also be seen as landmark join [13, 18]. That is, a join over a window with a fixed start (the time of query installation) and an open end.

In this chapter, we introduce the novel MultiStream operator which enables n-way join computation over data streams. With this operator in place, we look at constructing n-ary join trees from MultiStream operators and optimization of such trees. After that we discuss properties and limitations of the approach, and we present results of an experimental evaluation.

5.1 The MultiStream Operator

We start the explanation of the MultiStream operator with an example join query over three relations, $(R, S, T, \theta_{R,S}, \theta_{S,T})$. Tuples of each relation are placed in stores for each relation, labelled R-store, S-store, and T-store. When a new tuple arrives, it is sent to the store for its relation, and at the same time, it is sent to the other stores to produce a join result. A tuple $r \in R$ is sent to the R-store where it is placed in the prefix and awaits future probe tuples and at the same time it it sent to the S-store where the stored prefix is probed and the partial result $r \bowtie S = \{r \circ s \mid s \in S \wedge \theta_{R,S}(r, s)\}$ is computed. Unless $r \bowtie S$ is empty, it is sent to the T-store where T's prefix is probed and the final result for this tuple $r \bowtie S \bowtie T$ is produced.

It is unlikely to be beneficial to send tuples of R directly to T, as there is no predicate defined between R and T (analogous to the avoidance of computing Cartesian products in query execution for database systems). Thus, the mentioned order, sending R to S to T is the only considered way for tuples of R. For tuples of S, however, there are two options: First, sending $s \in S$ to R for computing $s \bowtie R$ using predicate $\theta_{R,S}$ and then to T for computing $s \bowtie R \bowtie T$. Or second, computing $s \bowtie T$ with help of $\theta_{S,T}$ and then $s \bowtie T \bowtie R$. Both cases can be used to produce a correct join result.

In general, given a query $q = (R_1, \dots R_n, \theta)$, a **MultiStream** operator consists of the **stores** for R_1 to R_n and **probe orders** σ_{R_1} to σ_{R_n} that compute q.

The probe orders define the strategy of routing tuples through stores to incrementally compute the join result. More formally, for tuples of relation R_i, we write the probe order as

$$\sigma_{R_i} := \langle R_{\sigma_i(1)}, R_{\sigma_i(2)}, \dots, R_{\sigma_i(n)} \rangle. \tag{5.1}$$

It is a permutation of the input relations of the query with $R_{\sigma_i(1)} = R_i$. A tuple $r_i \in R_i$ is sent to the $R_{\sigma_i(2)}$-store for computing $r_i \ltimes R_{\sigma_i(2)}$, this intermediate result is sent to $R_{\sigma_i(3)}$ for computing $(r_i \ltimes R_{\sigma_i(2)}) \ltimes R_{\sigma_i(3)}$, and so on. In fact, this so-called **probe order** is exploited to optimize network traffic. In the example above, we have probe orders $\sigma_R = \langle R, S, T \rangle$ and $\sigma_T = \langle T, S, R \rangle$ and either $\sigma_S = \langle S, R, T \rangle$ or $\sigma_S = \langle S, T, R \rangle$.

5.1.1 Scaling out

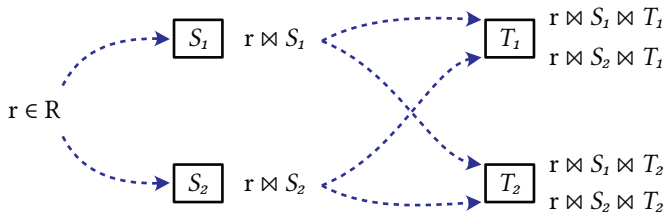
So far, the MultiStream operator was distributed to different stores, which is solely depending on the set of input relations of the query. Since stores are distributed components, instead of running on a single node it can be *scaled out* to multiple tasks on different compute nodes. For this, the stored prefix is partitioned to all these tasks and tuples are placed randomly.

To guarantee correctness, tuples are sent along probe edges and thus are broadcast to each task of a partitioned store, so all potential join partners eventually met. On one hand, a higher number of bytes is transferred through the network due to these broadcast copies. On the other hand, arriving tuples are probed against all partitions in parallel, which decreases overall latency.

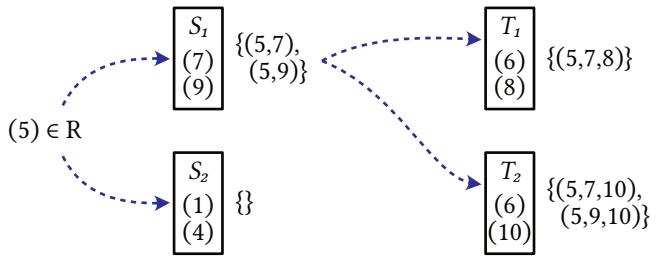
Example

For a more concrete look at the join procedure of MultiStream, let us consider a ternary join where the strategy is to send tuples from R first to the S-store and then to the T-store. Again, S and T are partitioned into two tasks. This scenario is in an abstract way depicted in Figure 5.1a. A tuple is sent in parallel to S_1 and S_2 where the intermediate join result $r \ltimes S$ is computed in two parts, $r \ltimes S_1$ and $r \ltimes S_2$. The former is sent to T_1 and T_2 and the latter is also sent to T_1 and T_2. The two tasks of T compute four partial join results whose union resembles the desired join result.

For a concrete example, let us consider that each tuple of R (S, T) only consists of a single value a (b, c) and the query wants all triplets where the value of R is less than the value of S is less than the value of T. Figure 5.1b shows the current state of the S and T tasks, e.g., task S_2 stores tuples (1) and (4) of relation S. Tuple $(5) \in R$ is arriving and sent to both tasks S_1 and S_2. While in S_1 both tuples satisfy the predicate $R.a < S.b$ and thus the result contains $\{(5, 7), (5, 9)\}$, no result is produced in S_2 und



(a) Routing installed for the operator.



(b) Concrete data flow for tuple $r = (5)$

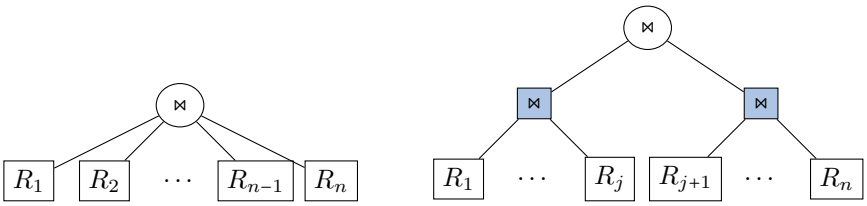
Figure 5.1: Join computation for query $q = (R, S, T, R.a < S.b, S.b < T.c)$.

thus no intermediate tuples need to be sent from S_2. The intermediate results from S_1 are then sent to T_1 and T_2. At T_1 only one join partner is found (only $7 < 8$) and at T_2, both arriving tuples can be joined with (10).

This example illustrates the difference between messages and tuples. Between S_1 and the T-tasks, only one message is sent which contains two tuples. Each message contains at least one tuple; an empty intermediate result does not generate a message as seen in S_2. The maximal size of a message is the size of the prefix stored in the sending task. If necessary, messages can be chunked without compromising the correctness of the join result.

5.2 Constructing Multi-Way Trees

There is no fundamental limit to the number of inputs a single MultiStream operator can handle. But instead of using a single operator, we can combine multiple MultiStream operators into a join tree. As MultiStream produces a streamed relation, this can serve as input to upstream operators. Such a join tree is depicted as a rooted n-



(a) Single MultiStream operator (b) Sequence of MultiStream operators

Figure 5.2: Two options for joining n relations using either a single MultiStream operator or combining multiple MultiStream operators into a join tree.

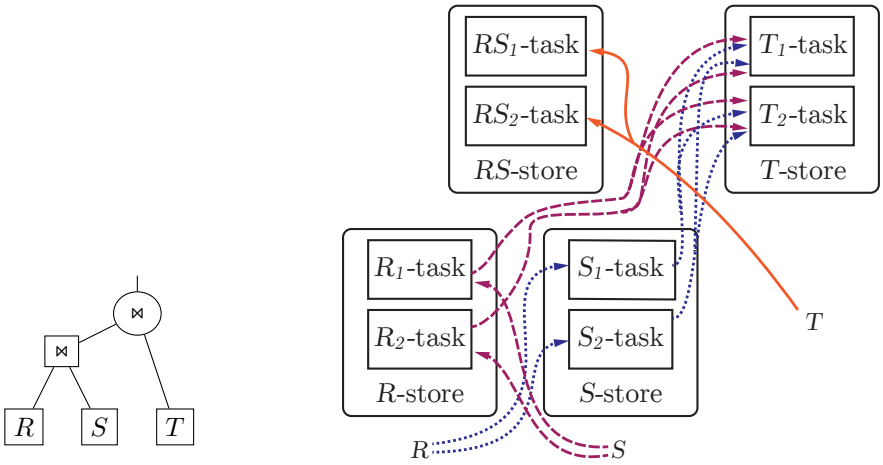
way tree, where each operator is an inner node or the root, and the sources are the leaves. In graphical notation, we depict all nodes that represent a materialized relation in a boxed style while nodes drawn inside of an ellipse (in this case, only the root) are not materialized.

Figure 5.2 illustrates how a join over n inputs can be computed by a single or a combination of three MultiStream operators. In Figure 5.2a all n inputs flow into a single MultiStream operator. As we discussed in the previous section, there are n stores, and n probe orders. In Figure 5.2b, the inputs are split into two partitions, R_1 to R_j and R_{j+1} to R_n. Both partitions are handled by a MultiStream operator each, and these operators are input to the root operator. Here, there are two additional stores required for materializing the intermediate results of $\bowtie_{i=1..j} R_i$ and $\bowtie_{i=j+1..n} R_i$, as well as probe orders between those two relations.

5.2.1 Inner Nodes as Communication Barriers

When a query is answered using a non-flat join tree, inner nodes of the tree act as barriers for communication. The iterative probing process is only conducted between all direct children of a node and if multiple descendants are subsumed into a single child node, only this child node needs to be probed.

Consider the join depicted in Figure 5.3a where the subquery over R and S is materialized and T is joined with $R \bowtie S$. Figure 5.3b illustrates the routing of probe messages in the physical graph generated for this plan. Tuples from R and S are still sent through both other stores to compute join results: First, they use the probe orders for the inner MultiStream operator, and then the one for the root operator. Tuples from T, however, are sent directly to the RS-store that hosts tuples of $R \bowtie S$. This also implies, that for R and S there is less freedom in choosing the probe orders: If R was first sent to the T-store for probing, this would not create the required result for $R \bowtie S$.



(a) Materialized join as communication barrier

(b) Routing

Figure 5.3: Three-way join with materialization of intermediate results.

5.2.2 Complexity of Multi-Way Trees

Figure 5.4 shows all five (structurally) different join trees over four relations. Consider Shape III, there we could exchange S_1 and S_3 and end up with different materialized intermediate results, and thus a different join plan. The MultiStream operator is commutative by construction, and thus, swapping S_1 with S_2 in Shape III would not lead to a different plan. The same is true for MultiStream operators on intermediate results, so swapping both, S_1 and S_3, and S_2 and S_4 essentially changes positions of the intermediate results in Shape III, thus leading to the same plan. This also means, that there is only one plan for Shape IV.

For a query over n relations, the number of different plans that can be constructed grows quickly. To see how fast, consider the choices for the root of such a plan: Either this root has n children with arities $1+1+\cdots+1$ (i.e., all children input relations), or it has $n-1$ children with arities $2+1+\cdots+1$ (i.e., a binary join and the remainder are input relations), or $3+2+1+\cdots+1$ (i.e., a ternary, a binary join, and the remaining input relations), and so on. The combinations of numbers that sum up to n is also called *partition* and is a lower bound for the number of plans. Asymptotically, the number of partitions grows $p(n) \sim (4n\sqrt{3})^{-1} \exp(\pi\sqrt{2n/3})$ [38]. For bigger numbers of relations ($n \geq 12$) the estimation from Section 2.4 for binary trees to grow with $C(n-1)$ already provides a higher lower bound. This suggests that the variations due to commutativity, which is not covered by $p(n)$, has a strong effect on the available plans.

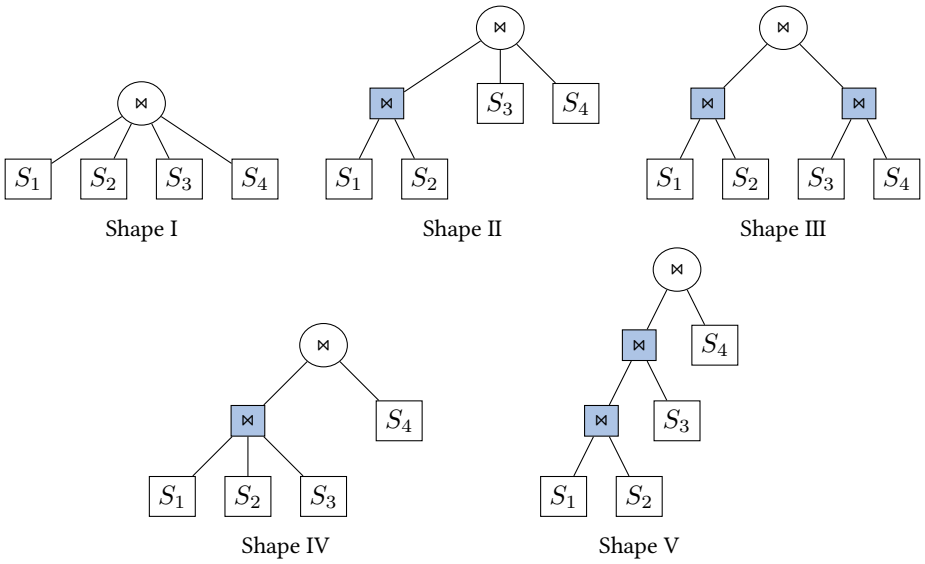


Figure 5.4: All *structurally* different join trees over four streamed relations.

5.3 Static Optimization

We have seen how a single MultiStream operator uses iterative probing to compute join results and how we can arrange multiple operators in an operator tree where inner nodes represent communication barriers but also require storage for the materialized tuples of these partial results. Naturally, this brings up the question of how to select a tree for answering a query at hand. Therefore, we now introduce a cost model and thereon we propose different optimization methods for trees of MultiStream operators.

5.3.1 Cost Model

Our cost model involves three aspects: **storage**, **communication**, and **latency**. The aspects describe characteristics of a deployed topology, however, we formulate them for operator trees which are easier to reason about.

Storage Cost

Storage is the amount of data that is stored across the topology. It consists of the sum of stored inputs as well as all intermediate results. For a tree $\mathcal{T}$ composed of a root O_{root}, inner operators $\mathcal{O}$, and relations R_i, we define storage cost SCost formally as:

$$\text{SCost}(\mathcal{T}) = \sum_i |R_i| + \sum_{O \in \mathcal{O}} |O|. \tag{5.2}$$

The root is not influencing the storage cost, as it does not need to be materialized. For inner nodes $O \in \mathcal{O}$ wich materialize intermediate relation $(R_1, \dots R_k, \theta)$, we employ a standard cardinality estimation for join sizes [27]:

$$|O| = \prod_{i \in 1..k} |R_i| \prod_{i,j \in 1..k, i \neq j} f_{i,j}. \tag{5.3}$$

Probe Cost

The communication cost is composed of **probe cost** and **storage cost**. Storage cost appears also under communication, as each tuple that is stored must be sent to the storage location. And since each intermediate result is created exactly once, it is sent once to its assigned store. The probe cost of an entire tree is composed of the individual probe cost of each involved operator:

$$\text{PCost}(\mathcal{T}) = \sum_{O \in \mathcal{O}} \text{PCost}(O). \tag{5.4}$$

The probe cost of an individual operator is more involved. We start with a single probe order for R_i. At first, we only look at the sizes of the produced intermediate results. All tuples from R_i are broadcast to the $R_{\sigma_i(2)}$-store in order to be joined there. The expected result contains $|R_i| \cdot |R_{\sigma_i(2)}| \cdot f_{i,\sigma_i(2)} \cdot \frac{1}{2} =: c_2$ tuples. This estimate is grounded on standard selectivity estimation and the fact that only tuples $r_{\sigma_i(2)} \preccurlyeq r_i$ are contained in the join result. The generated results are then sent to the $R_{\sigma_i(3)}$-store, and here we expect $c_2 \cdot |R_{\sigma_i(3)}| \cdot f_{i,\sigma_i(3)} \cdot f_{\sigma_i(2),\sigma_i(3)} \cdot \frac{1}{3}$ tuples. Again, this is a combination of the selectivity-based estimation of the join size combined with the restriction on tuples with $r_{\sigma_i(3)} \preccurlyeq r_i$. With these considerations, and the fact that each tuple created at step $j-1$ is broadcast and thus copied N_j times, the general estimation on probe cost for the MultiStream operator O is:

$$\text{PCost}(O) = \sum_{2 \leq j \leq n} N_{\sigma_i(j)} \cdot \left| \ltimes_{k=1}^{j-1} R_{\sigma_i(k)} \right| \cdot \frac{1}{j}. \tag{5.5}$$

Communication cost considers the number of tuples that have to be forwarded to the individual components. This number is heavily dependent on the degree of parallelism of components due to broadcast tuples. A high number of sent tuples implies not only a higher bandwidth consumption but also more logic to be executed at the receiving nodes, thus, reducing this factor will enable higher overall throughput.

Latency

Latency is the average expected number of network hops a tuple undergoes until a join result can be returned and is important for real time requirements. For a tree

where input relation R_i is connected to the root O_{root} through inner operators $\mathcal{O}$, the expected number of hops is:

$$\text{ELat}(\mathcal{T}) = \sum_{O \in \mathcal{O} \cup \{O_{root}\}} out(O) - 1. \tag{5.6}$$

Each operator is completed from the input by probing the other children of this operator, hence $out(O) - 1$. We currently ignore the local evaluation cost of the join for this cost estimation, which means that if expensive predicates are involved, e.g., for image analysis [63], the estimation has to be further refined.

For this chapter, we assume data characteristics like arrival rate of streams and selectivities of the join predicates are known upfront and are assumed to be stable, such that the plan for a query is not changed during query time (hence the section title "Static Optimization"). We describe dynamic re-evaluation later in Chapter 7.

5.3.2 Locally Optimizing MultiStream

While technically every set of permutations could serve as a probe order in a MultiStream operator and produce the correct result, it makes sense to select one that does not generate too much network overhead according to the estimation of Formula 5.5.

Due to the similarity to textbook join-order optimization, we could proceed by ordering the relations according to the size of the join result similar to the approach by Golab and Özsu [30]. However, we also have to incorporate the parallelism of the stores as exemplified in the following: Consider a join $R \bowtie S \bowtie T$ with $|R| = 2\,000$, $|R \bowtie S| = 1\,000$, $|R \bowtie T| = 1\,500$, and the parallelism of the stores S and T is $N_S = 5$ and $N_T = 1$, respectively. If we decide to route tuples from R first to S, then to T, it would cause fewer *different* intermediate tuples to be created, than sending the tuples first to T and then to S. However, due to the parallelism of S, *more copies of the same tuple* are created: while $\langle R, S, T \rangle$ provokes 10 000 + 1 000 tuples to be communicated, $\langle R, T, S \rangle$ only needs 2 000 + 7 500 as given by the inner sum of Formula 5.5.

Greedy Probe Order Ordering

The entire space of possible probe orders for n relations consists of $(n-1)!$ possibilities to choose from, and we need one for each starting relation. In order to avoid scanning in total $n!$ permutations and still find a set of probe orders with decent PCost, we leverage the bottom-up greedy algorithm shown in Algorithm 1. This algorithm exploits information about the parallelism of the stores and, in Line 3, it iterates over all pairs of to-be-scheduled relations (R_j, R_k) for finding the next relation. With this approach, we can determine the next relation R_j based on the number of tuples that have to be sent to this relation, $|\sigma_i| \cdot N_j$. We use σ_i as shorthand for all already scheduled relations. At the same time, we avoid too big intermediate results by computing

Algorithm 1 Greedy algorithm for probe-order optimization.

input: starting relation R_i
other relations R (without $\{R_i\}$)
parallelism of stores N

1 $\sigma_i := \langle R_i \rangle$
2 **while** $R \neq \{\}$
3 $R_j, \_ := \arg\min_{R_j, R_k \in R} |\sigma_i| \cdot N_j + |\bowtie(\sigma_i \cup R_j)| \cdot N_k$
4 $R = R \setminus \{R_j\}$
5 $\sigma_i = \sigma_i \circ \langle R_j \rangle$
6 **return** σ_i

the possible number of to-be-sent relations in the next iteration. Note that R_k is not scheduled yet, rather all possible R_k are tested in order to avoid a choice for R_j that causes a bigger intermediate result in the next iteration.

This algorithm is invoked once for each starting relation of the MultiStream operator at hand, thus, the overall selection for n input streams takes $O(n^4)$ time.

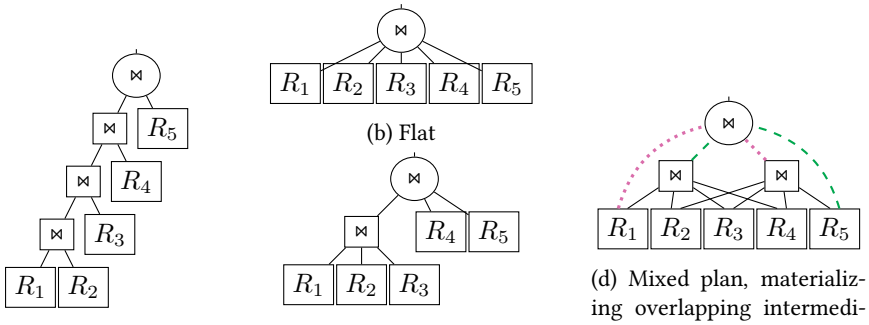
5.3.3 Global Join-Tree Optimization

Construction of trees poses the decision how much storage we want to spend or how much communication we want so save. For real-world applications, the ultimate optimization target is depending on the situation, and can be a mixture of the aforementioned costs as well as additional constraints, like:

- for a certain system capacity, find a plan that maximizes usage of that capacity and thereby minimizes the total probe cost,
- for a certain system capacity, find a plan with minimal average depth, or
- find a plan with minimal storage cost.

If the topmost priority is saving the space required for storing prefixes, then a trivial solution is to just not materialize anything else than the inputs, as shown in the flat tree in Figure 5.5b. There, $R_1, \ldots, R_5$ are joined using a single MultiStream operator.

One constructive optimization approach is to just build a left-deep tree like the one in Figure 5.5a. Here, the intermediate result of $R_1 \bowtie R_2 =: R_{12}$ is materialized, the result of $R_{12} \bowtie R_3$ is computed and materialized, and so on. In contrast to the join ordering in relational database systems, where allocated memory can be freed after an operation is completed, we continuously read fresh input. Thus, the intermediate results are never "finished" and the tuples have to be kept forever in the case of full-history joins. As already hinted at in Chapter 1, a left-deep plan might be prohibitively



(a) Left-deep

(b) Flat

(c) Mixed plan

(d) Mixed plan, materializing overlapping intermediate relations

Figure 5.5: Different possibilities of constructing query plans for five input relations.

expensive, as it requires $n - 2$ additional materialization points for a query over n relations.

To avoid hitting such a limit, a plan like in Figure 5.5c can be used. Consider the case of a join where each pair of relations R_1, R_2, and R_3 produces a big result, but the result of the join $R_1 \bowtie R_2 \bowtie R_3$ is just roughly of the same size as the input relations. Then, it is preferable to materialize only the entire result instead of the intermediate joins. Figure 5.5c is also an example for the role of materialization as boundary for communication: When tuples of R_4 arrive, they only have to be probed against the prefix of R_5 and the prefix of the materialized result of $R_1 \bowtie R_2 \bowtie R_3$, but not against R_1, R_2, or R_3 individually. Each of the latter relations is still probed against all other relations.

As potential extension, Figure 5.5d shows a plan which materializes intermediate results from overlapping intermediate relations, which are $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ and $R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$. This way, tuples from both, R_1 and R_5 can compute the result in a single hop (compared to the left-deep plan where only R_5 can do that). Obviously, both intermediate results have to be maintained. We will not investigate such overlapping strategies in this thesis.

Greedy Algorithm for Constructing Generalized Left-deep Query Plans

Here, we limit the output of the tree optimization to generalized left-deep trees. A generalized left-deep tree is a tree with n-ary nodes where at most one child is a left-deep tree. To avoid enumerating all $n!$ possible trees, we use a greedy heuristic, shown in Figure 5.6. The algorithm generates a sequence of relations that are interpreted as leaf nodes of the left-deep tree. In order to avoid a bad start, each relation R_i is considered a starting relation. This resembles a classical, textbook approach for generating left-deep trees. In variable l the sequence of the relations is kept, r indicates the remaining relations, and b tracks the budget spent so-far. However, each left-deep plan

```
input: relations R, budget B
1  for R_i ∈ R
2      l ← [R_i]; r ← R ∖ {R_i}; b ← Σ_{R_j ∈ R} |R_j|
3      while r ≠ {}
4          R' ← arg min_{R_j ∈ r} |l ⋈ R_j|
5          b ← b + |l ⋈ R'|
6          if b > B
7              break
8          r ← r ∖ {R'}
9          l ← l ∘ R'
10     updateBest(l, b)
```

Figure 5.6: Greedy algorithm for generalized left-deep plan construction under budget constraints.

over n relations consists of $n-2$ materialized intermediate results, which might exceed the available budget. Thus, we add the option to terminate the construction of the tree early, such that the final plan is composed of a k-way MSJ root where one child is a left-deep subtree. This subtree is as big as the budget permits.

As each relation's prefix needs to be stored, the budget spent is initialized with the sum of the prefix sizes in Line 2. As long as there are remaining relations in r, the algorithm tries to schedule another relation R' to l and chooses the one with minimum join size (Line 4). If this choice would lead to a plan exceeding the budget B, it is not included and the left-deep part with R_i is complete, as all other possible choices would also exceed the budget. Otherwise, R' is removed from the remaining relations and added at the end of l.

Restriction to Linear Join Graphs (Chains)

In Algorithm 5.6, we limited the form of the output tree upfront to be left-deep. In order to support large join graphs, here, we shrink the search space by limiting the exploration to only linear join graphs (also called chains). If multiple linear join graphs exists, e.g., in a clique, a random one is selected. This means, the optimization strategy internally considers only joins of relations $R_1, ..., R_n$ with join predicates $\theta_{i,i+1}$.

The strategy operates in a top-down fashion and starts with a flat query plan consisting of a single MultiStream operator. Then, it proceeds with iteratively merging children into new operators, until the system capacity is reached. The pseudo code in Algorithm 2 describes this procedure. First, a representation of the flat tree is initialized in Line 1 as list of lists and the initial budget is set as for the previous approach to generate left-deep trees. As long as the budget is not exceeded, the algorithm searches for relations to merge, as follows: For all indexes i and j over the flattened list T, the ones which reside in the same sublist are considered valid. Then the pair (i, j) with

Algorithm 2 Top-down strategy for incrementally adding materialization.

input: relations R, budget B
1 $T \leftarrow R.map(\lambda x.[x])$
2 $b \leftarrow \sum_{R_j \epsilon R} |R_j|$
3 **while** $b < B$
4 $\quad (i,j) \leftarrow argmin_{(i,j),s.t.valid(T,i,j)} SCost(mat(T,i,j))$
5 $\quad$ **if** $B > SCost(mat(T,i,j)$
6 $\quad\quad$ **break**
7 $\quad T \leftarrow mat(T,i,j)$
8 **return** convertToTree T

minimal cost for materializing the join of relations $R_i, \ldots R_j$ is selected in Line 4. If the cost for materializing this would exceed the budget, the algorithm terminates—otherwise, it introduces a materialization for these relations and continues. Finally, the nested list of lists is converted into a tree and returned in Line 8. The runtime complexity of this algorithm is in $\mathcal{O}(n^4)$.

Iterative Merging Strategy

We put forward another approach, seen in Algorithm 3: Given a join over n relations, we start with a single n-ary node combining all relations (Line 1). Then we re-iterate (Line 2, loose syntax for brevity) as long as we find a node N with a proper subset of children S that consists of at least two leaves and the communication and storage cost, as computed in Formulae 5.4 and 5.2, for S is between given bounds L and U. If we find such N and S, then we replace in N these children by a freshly created node that contains the children (Line 3–Line 4). Thus, materialization stages are only introduced if they do not occupy too much storage ($< U$) and if they seem useful at all ($> L$). These bounds can be set according to the memory one wants to invest. A higher value for U allows creating plans that materialize bigger intermediate results, while a higher value for L restricts creating more employed worker nodes (bolts) that would only save little communication. These parameters can be set depending on to the sizes of the input relations, e.g., $L = max(\mathcal{S})/2$ and $U = 2max(\mathcal{S})$.

This algorithm reflects this tradeoff by starting with a tree that has minimal storage requirements and iteratively adds new materialization points that worsen the storage cost of the resulting tree. At the same time, the communication cost is lowered, as intermediate results do not have to be reconstructed by probing.

Algorithm 3 Iterative construction of join trees.

input: relations $\mathcal{S}$, bounds L, U
output: join tree $\mathcal{T}$
1 $\mathcal{T}$:= createNode($\mathcal{S}$)
2 **while** $\exists N \in \mathcal{T}.\exists S \subset N.\ \text{card}(S) \geq 2 \swarrow$
$\wedge L < \text{CCost}(S) \wedge < \text{SCost}(S) < U$
3 X := createNode(S)
4 replaceChildren(N, S, X)
5 **return** $\mathcal{T}$

5.4 Correctness

In this section, we discuss two aspects of correctness for our approach. First, exactly-once processing, which makes sure that the result contains all the expected tuples and only them. And second, we discuss ways to maintain correctness under different system failures.

5.4.1 Exactly-Once Processing

In order to ensure that the join is computed correctly, i.e., each element from the crossproduct of all involved relation that satisfies the join predicate is output exactly once, we need to show two properties: each result is computed at least once, and each result is computed at most once. As basis for a formal reasoning and similar to [47], we introduce a total order on the tuples

$$\forall r_i \in R_i, r_j \in R_j.\ r_i \prec r_j \vee r_j \prec r_i \tag{5.7}$$

where R_i and R_j are relations from the query and $\prec$ is any order, e.g., the lexicographical order over timestamp and relation name.

We now decompose the desired result for a join $\mathcal{R}$, into all subsets where a tuple of stream relation R_j arrived last:

$$\begin{aligned}\mathcal{R} &= \{r_1 \circ \cdots \circ r_k \mid r_i \in R_i \wedge \theta \text{ is satisfied}\} \\ &= \bigcup_{1 \leq j \leq k} \{r_1 \circ \cdots \circ r_k \mid r_i \in R_i \wedge \forall_{j' \neq j} r_{j'} \prec r_j \wedge \theta(r_1, .., r_k)\}\end{aligned}$$

This decomposition is correct, as there is a total order among the tuples, thus, each result $r_1 \circ \cdots \circ r_k$ falls in one of these partitions. That means, it is enough for each tuple r_j to be joined with all tuples that have a lower timestamp, or, expressed more intuitively, with all tuples that arrived earlier.

Consider an arbitrarily chosen tuple $r = r_1 \circ \cdots \circ r_n$ of the join result. There is one tuple $r_i \in \{r_1, \ldots, r_n\} =: \tilde{r}$ such that $\forall r_j \in \tilde{r}.\ r_j \prec r_i$. Intuitively, r_i is the tuple that

arrived most recently at a MultiStream operator and it visits the stores according to probe order σ_{R_i}.

Lemma 1, MultiStream computes each join result at-most-once. Proof: We make sure that no other probe order than σ_{R_i} can produce r by adding $\prec$ to the join predicate during probing. When some r_j is probed against the prefix of a store R_k, all resulting tuples satisfy $\theta_{j,k}(r_j, r_k) \wedge r_j \prec r_k$. The resulting intermediate tuple is assigned the higher timestamp, thus, during iterative probing, only tuples with timestamp lower than the initial tuple can be joined. □

Lemma 2, MultiStream computes each join result at-least-once Proof: r_i is sent as probe to the other stores in order to probe with tuples $r_j \prec r_i$. It might occur that a tuple r'_j with $r_j \prec r'_j \prec r_i$ arrives *after* r_i. Logically, r_i had to be probed against r'_j but this was physically impossible when the store received r_i. In order to eliminate this source of error, a so called probe log is used. Each arriving probe tuple is kept in the probe log and when a store tuple arrives, the probe log is checked for such a missed probe tuple. Compared to the stored prefix, the probe log can be kept small using stream punctuation techniques. □

Based on the two lemmata, we can directly deduce the following theorem.
Theorem: As each join result is produced at-most-once and at-least-once, the Multi-Stream operator produces overall the correct results.

Example

Consider the three-way MultiStream operator computing (R, S, T, θ) with local join orders $\langle R, S, T\rangle$, $\langle S, R, T\rangle$, and $\langle T, S, R\rangle$. This operator is realized with a store for each relation and the interconnection between these stores. Further, assume that each store has stored the tuples as indicated in Figure 5.7, where the indices indicate the timestamps of the tuples, and let all tuples satisfy the join predicate. Here, tuple r_8 just arrives and is sent to the R-store for storing and to the S-store for probing. There the intermediate result $\{r_8 \bowtie \{s_2, s_5\}\}$ is computed which is again sent to the T-store for probing. It is crucial that here only t_4 is considered for probing. If said intermediate result would also be joined with t_9, then the T-store would report $\{r_8 \bowtie \{s_2, s_5\} \bowtie \{t_4, t_9\}\}$. At the same time, we only know that tuple t_9 is stored but it could be still on its route as instructed by the operator and then R could produce the result $\{t_9 \bowtie \{s_2, s_5\} \bowtie \{r_1, r_3, r_8\}\}$, as r_8 in the meantime arrived at the R-store. Looking at these two results, the tuples (r_8, s_2, t_9) and (r_8, s_5, t_9) occur in both presented results, clearly violating the at-most once guarantee.

We solve this problem by assigning each source tuple a timestamp and join this tuple only with other tuples having a smaller timestamp. This means, that the source tuple's timestamp is kept at each network hop independently of the timestamps of other tuples in the intermediate results. The dispatcher assigns timestamps as mono-

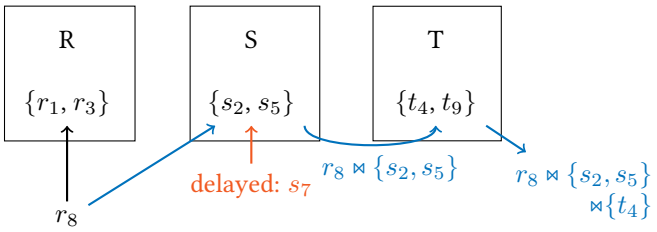


Figure 5.7: Timestamp-sensitive routing of three-way MultiStream operator.

tonically increasing sequence numbers. If for some reason multiple tasks for the dispatcher are needed, then the dispatcher task's id is added to the timestamp and serves as tie breaker.

The next question is, how to deal with delayed tuples. Consider again the example in Figure 5.7 after r_8 is fully processed. Now, tuple s_7 arrives late at the S-store. According to our approach, if this tuple arrives at the R-store, regardless of whether r_8 is already stored there or not, s_7 and r_8 cannot be joined there due to the timestamps. Instead, when r_8 arrives at the S-store it is put in a special buffer which is inquired every time a tuple of S arrives for storing. Then, the arrival of s_7 lets not only s_7 be stored, but also s_7 is probed against this buffer and the intermediate result $r_8 \bowtie s_7$ is produced and sent to the T-store. In order to prevent this special buffer from occupying too much heap space, tuples which will not have a delayed join partner have to be discarded. This can be achieved by stream punctuation techniques, which send special packages with the current maximum timestamp through all streams. Since all components are connected via FIFO streams, the arrival of a punctuation package with timestamp t indicates that no other tuples with timestamp $t' < t$ can arrive later. Hence, the special buffer can be freed from all tuples with such a lower timestamp.

5.4.2 Fault Tolerance

The problem raised by a node failure translates to the loss of a part of the state of the join operator. Consider the S-store of a join $R \bowtie S$ being distributed on machines S_1 and S_2. If S_2 dies, arriving tuples from R can only be sent to the remaining store and $R \bowtie S_1$ can be computed but not the entire result. A way to solve this problem would be to persist every incoming tuple on the worker node. Then, the process working for S_2 can be restarted and recover its state.

Another way, which also prevents the system from waiting until the restart is complete, is to back up each task by a secondary task that has a copy of the data of the primary [37]. This way, the total memory consumption of the plan would double, but only store messages would have to be sent twice, and in case of a failure the replicated node can take over the join processing. When the failed node restarts, it

can recover its state from the other replicated node leading to a (presumably) short peak in network usage.

A third way of recovering the state is possible if the source streams support replaying the data streams as, e.g., Apache Kafka does [9]. Then, the tuples from the relevant streams can be fed again into the system and need only to be routed such that the operator's state can be restored. However, due to the random nature of the partitions, all other tasks' states have to be restored as well.

5.5 Experiments

In this section, we explore the effects of different choices of operator trees on achievable performance of the overall join computation workload. We deploy the Storm topologies generated by CLASH on an Amazon EC2 cluster consisting of multiple t2.large instances each having two virtual cores, 8 GB main memory, and being interconnected with "low to moderate" networking performance. According to iperf measurements the network provided at least 50 Mbps. Each instance is running up to two workers and the number of instances is adjusted based on requirements of the plan and degree of parallelism. For example, for a plan that requires 5 stores and each store has a parallelism of 3, we would allocate $\lfloor 5 \cdot 3/2 \rfloor$ instances. Storm version 1.1.0 is used running on Oracle Java 1.8 and Ubuntu Server 16.04. Locally, we use a plain nested-loops join which is able to produce the desired result for any given computable binary predicate.

5.5.1 Setup

We use data from the TPC-H benchmark [3], which is commonly used for evaluating stream processors [47, 61]. We generate the data set with scale factor 1 and the joins inside queries Q2, Q3, and Q5. For example, Q2 actually filters according to a certain `part` size and type and selects the `supplier` that has minimal `cost`. However, we are only evaluating the join of the base relations without considering nested subqueries or aggregations. The TPC-H queries consist of equi joins according to the foreign-key relations between tables. However, we refrain from exploiting this in the routing. This means that our performance results remain valid for other, non-equi-join predicates.

Further, we use a custom generated dataset that allows the creation of linear join queries with arbitrary selectivities for the individual joins. This way we can specifically explore the effects of the intermediate result sizes on the performance of different plans. We appoint the queries according to the combination of low (L), medium (M), or high (H) selectivities, e.g., as Custom-*MHL* for a four-way join chain with selectivities $f_{1,2} = L$, $f_{2,3} = M$, and $f_{3,4} = H$, and explain the choices for L, M, and H as

well as the sizes of the relations where necessary.

We let the topology consume the datasets from Kafka, one topic for each relation. This enabled repeating each experiment while having an environment close to an industrial production setting. Specialized installations can go further and partition the topics and read in parallel, however this goes against our core idea of enabling users to pose arbitrary queries.

When a Kafka topic is read multiple times, each but the first reading are as fast as supported by Kafka or the consumer. For rate-limited experiments, like the latency experiment in this section, or later experiments for windows, we use a driver program which acts as Kafka producer. This program feeds given lists of tuples into Kafka and it can configure a rate for each of the lists.

We compare the following approaches, respectively, shapes of the join trees:

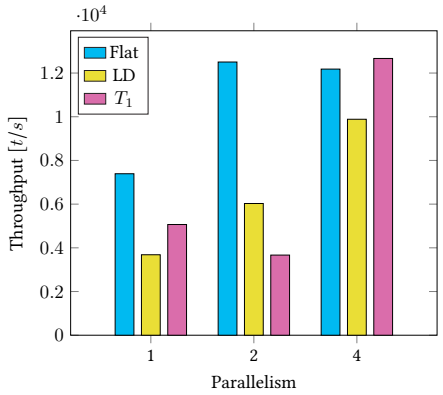
- A plan that only consists of a single non-materialized MultiStream operator, with all involved relations as leaves, denoted **Flat**.
- A plan consisting of a left-deep combination of binary MultiStream operators. This resembles the BiStream approach as suggested in [47], denoted **LD**.
- For more than three relations, there are other plans possible than the ones listed. These are denoted T_i and explained later.

While the system is able to incorporate other join operators like HyperCube, we do not report on results, since it is already outperformed by BiStream, as reported in [47]. In fact, it was deemed not applicable to joins involving more than two relations in first experiments.

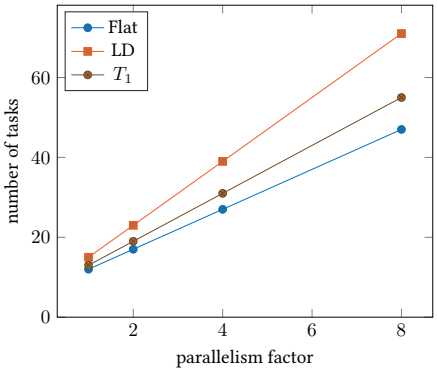
5.5.2 Throughput and Scalability

For measuring the throughput of different plans, we feed the data into the topology as fast as possible and take the time difference between the first read tuple by the dispatcher and the last output join result. The throughput is then the number of input tuples divided by this time difference. The results for Q2 are shown in Figure 5.8a; here T_1 is an optimized plan. We see that with parallelism 1, the flat plan provides the highest throughput, which is presumably because there is less overhead due to fewer active bolts in the topology.

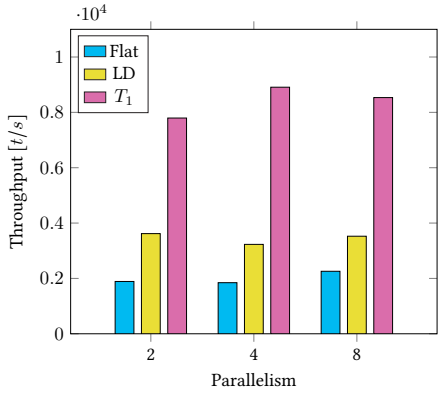
In order to see how the throughput behaves when a store is scaled over multiple Storm tasks, we globally increase the degrees of parallelism, i.e., the number of Storm tasks that run a certain store bolt. This means, with a higher degree of parallelism, the number of tasks grows depending on the number of bolts. Figure 5.8b illustrates that the deployed Storm topology consists of more concurrently active tasks for the left-deep plan than the other two plans with the same degree of parallelism and that



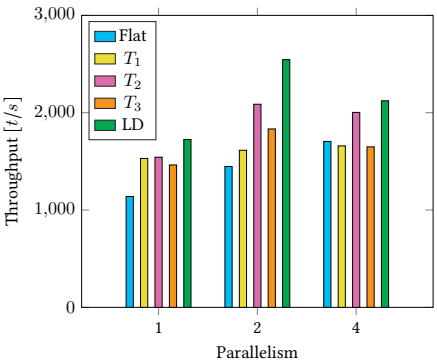
(a) Throughput of different plans for global scaling (Q2).



(b) Impact of parallelism factor on the number of instances.



(c) Throughput for storage sensitive scaling (Q2).



(d) Throughput of different plans for global scaling (MHM).

Figure 5.8: Throughput and parallelism.

the number of tasks grows linearly. Figure 5.8a shows the behavior of the throughput for different plans and varying parallelism: With higher parallelism, the left-deep BiStream plan scales better because it requires less tuples to be sent around during join computation.

It is not necessary to scale globally, but specifically increase the number of tasks for that store which needs to keep the most tuples in memory. This might be necessary if the desired prefix gets too large, and thus a scale out cannot be avoided. The effect of this can be seen in Figure 5.8c where for answering Q2 the parallelism of the 'partsupp'-store was increased. Neither of the queries significantly gain or lose in terms of throughput which is important, because it confirms, that a specific scale-out operation can be done without harming the performance.

The selectivities of joins in Q2 make intermediate relations smaller than input relations. In contrast, MHM is designed such that the intermediate relations are larger than the input relations. In Figure 5.8d we see the impact of this, most importantly the

throughput is lower than for Q2 by an order of magnitude. Secondly, increasing the parallelism does not bring a significant performance boost with any plan. The plans used correspond to the shapes illustrated in Figure 5.4, the flat plan matches shape I and the left-deep plan is an instance of shape V. In plan T_1, $R_1 \bowtie R_2$ is computed by a MultiStream operator and a MultiStream root combines this result with R_3 and R_4 (shape II), plan T_2 materializes the results of $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ yielding a balanced tree (shape III), and plan T_3 combines a R_1, R_2, and R_3 in a three-way MultiStream operator (shape IV). Hence, for join queries that cause huge intermediate results they do not have to be stored in order to get throughput performance gains.

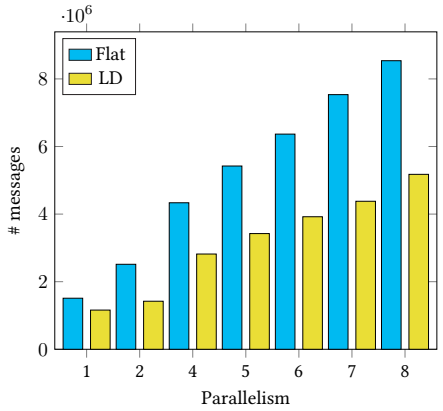
While communication is an important factor for scalability (i.e., the less communication the better the scalability) it also influences the variation in achievable throughput. If we measure the throughput of a workload, generally, a single run takes less than 10 minutes. However, from one to the next measurement the differences may be huge, which is especially true for communication intensive plans. We expect such a measurement to result in significantly less scattering if conducted in a more controlled environment. However, this indicates what to anticipate when deploying such a plan in a shared environment like EC2. Still, such a setup is becoming more and more relevant in today's cloud computing landscape.

Scaling also affects the number of tuples sent during processing of a workload. Figure 5.9a shows how the number of observed communication actions changes when Q3 is processed using flat and left-deep plans with higher degree of global parallelism. As expected, the left-deep join plan causes significantly less communication which is saved by materialization and thus avoiding re-computation of intermediate results.

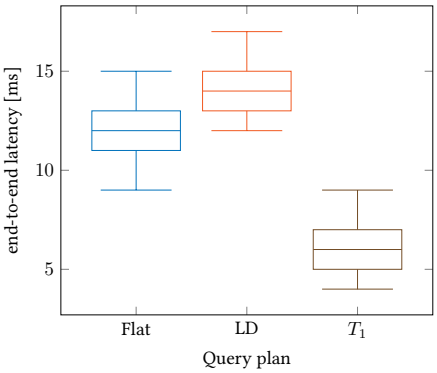
5.5.3 Effect of Materialized Intermediates on Communication

Here, we compare the savings for different scenarios. First, for custom-*LLLL* a five-way, low-selectivity join with equal relation sizes of 10^6 and a pairwise selectivity of 10^{-8}. Figure 5.10a shows the number of probed tuples depending on the capacity of a single store. If the task capacity is enough for a tenth of the incoming tuples of each relation (the leftmost case on the x-axis), over half of the probe tuples can be saved with **LF** compared to a **Flat**. The more task capacity there is, the less need of parallelizing the individual stores, and consequently also less probe overhead. If every task can handle the entire incoming relations, the difference is negligible (the rightmost case on the x-axis).

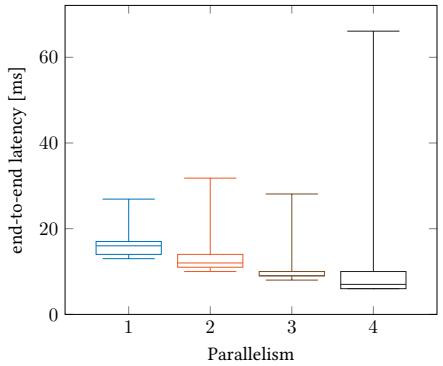
For medium selectivities of 10^{-6} as shown in Figure 5.10b, the left-deep plan still shows an advantage, however, the relative saving is not that big anymore. If the selectivities are getting larger to 10^{-5} , shown in Figure 5.10c, then **LD** becomes even worse. This effect is due to the increased parallelism requirement on the stores of the



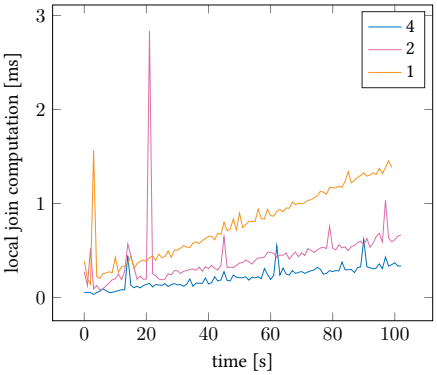
(a) Number of messages sent between stores.



(b) Latency for different plans computing Q2.



(c) Latency for a single plan, varying parallelism.



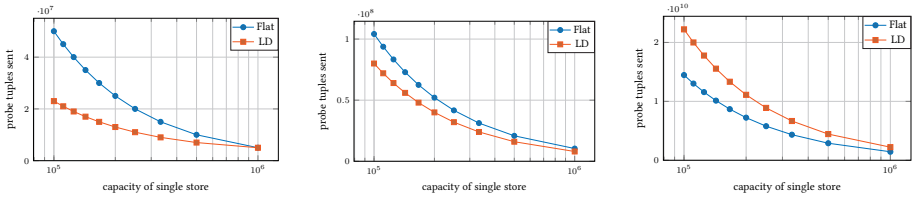
(d) Latency of local join computation per probe tuple.

Figure 5.9: Latency and communication behavior.

intermediate relations: the higher the selectivity, the more intermediate results, and the more tasks are required in order to store the prefixes. Therefore, probe tuples have to be broadcast to more stores, increasing the overall communication. This means, a single MultiStream operator is very well usable for scenarios where high selectivity joins are involved.

5.5.4 Latency

To measure the latencies of tuples, we assign each tuple t the timestamp of the system time using the Java system method `currentTimeMillis()`. The resulting tuple of a join between t and another previously stored tuple gets the same timestamp. If this tuple finally arrives at the sink, i.e., if t finds join partners such that the overall query is satisfied, the sink again reads the current system time and reports the difference between those timestamps in milliseconds. This measurement requires the clocks of the hosts where dispatcher and sink run to be synchronized, which we ac-



(a) Low selectivity. (b) Medium selectivity. (c) High selectivity.

Figure 5.10: Effect of task capacity on sent tuples for linear queries with varying selectivity.

complish by calling the Unix command `ntpdate` before every run. In contrast to the throughput measurements, here we feed the input tuples with a low rate into the topology in order to avoid measuring the time how long a tuple is buffered in the in- or output queues of the tasks.

The boxplots in Figure 5.9b show that intelligent placement of communication boundaries effectively limits the latency encountered for Q2, where the average latency of the mixed plan is half of the latency of the naive left-deep or flat plan. The boxes are showing the quartile bounds and the whiskers indicate the 2.5 and 97.5 percentiles. The reason behind this is the guaranteed low number of network hops between hosts. Figure 5.9c shows the results for answering query MMM under different parallelization factors. The median latency is reduced when using more store instances in parallel. However, also much more tuples need also significantly more time, since the network load increases due to the excess of broadcast tuples.

In order to understand the decrease in mean latency, we look at the time needed to compute a local join. As we use nested-loops joins in order to remain compatible with the ultimate goal of computing theta joins, the complexity depends on the size of the stored prefix. Thus with a parallelism factor of 4, only a quarter of the join predicates have to be evaluated in each store instance compared to parallelism factor 1, and the evaluation can be done in each store instance in parallel. Figure 5.9d shows how the time to locally compute a join when a probe tuple arrives changes over the course of a workload computation. As more tuples arrive, the time also changes. However, with a higher degree of parallelism, the number of join candidates found in a store instance per probe grows more slowly, hence the time of the local join computation influences the end-to-end latency less. Similar to the avoidance of broadcasting tuples, here, it is desirable to get more information about the join predicate in order to use a specialized join algorithm, e.g., a hash join for equi-join queries.

5.5.5 Storage Space Occupation

The overall storage space occupation of a topology can be statically computed if, as we assume, the relation sizes and join selectivities are known. The least storage is

occupied by a single flat plan over every relation, and the most (in terms of non-maliciously evil query plans) by a left-deep tree that needs to store $n-2$ intermediate results when joining n relations. Tuples are not stored redundantly when operators have a higher degree of parallelism, thus, scaling out does not influence the total amount of required space.

For Q2, the flat plan needs to store $|Q|+|PS|+|S|+|N|+|R|$ tuples, while the left-deep plan needs additionally $|N \bowtie R|+|N \bowtie R \bowtie S|+|N \bowtie R \bowtie S \bowtie PS|$ tuples. Q3 needs only a single additional store, and Q5 needs two additional stores for intermediate results when using a left-deep plan. As visualized in Figure 5.11a the left-deep join plan consumes for each query more space obviously for storing the additional tuples, for query Q2 by factor 1.82, for Q3 and Q5 about 1.2, and 1.5 for MMM and MLM.

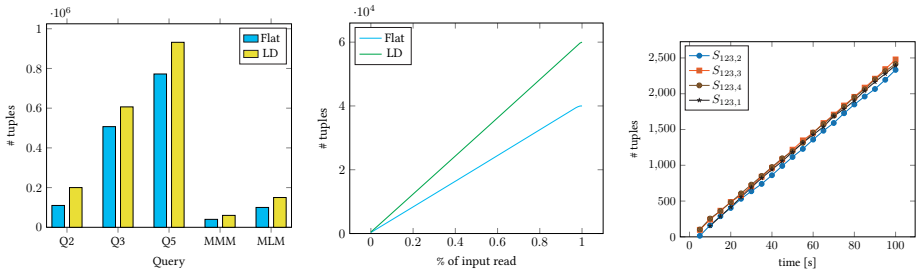
In Figure 5.11b, we show for query MMM how many tuples are kept in the heap during processing depending on how much percent of the data is read. The steeper slope of the left-deep plan's space usage originates from the additionally stored intermediate results in the MultiStream operators. With these observations about the storage requirements, it is clear that if the goal is to provide correct join results over large windows—or even for a full history—while using limited storage resources, we can resort to a storage-friendly plan as consisting of a single non-materializing root. Considering in Figure 5.11b that only 40k tuples can be held in memory at the same time, then, the left-deep plan would have to discard previous tuples after only two thirds of the workload being read.

For each materialized operator, the per-instance storage occupation is interesting as it is undesirable to have one instance that stores way more tuples than the others. The results shown in Figure 5.11c confirm that the usage of Storm's shuffle grouping evenly distributes the tuples to the instances of each store throughout the processing of a given workload. Here, the number of stored tuples is shown for four instances of the the same MultiStream operator's store. If the distribution would use a value-based partitioning, which is commonly used for equi-join computation, then additional measures have to be taken in order to avoid uneven resource usage due to data skew. The message distribution between the stores, however, depends on the plan used.

5.5.6 Message Load

We count the messages that are sent during join processing and compare the number of messages sent in different plans for different queries. As a reminder, a message may consist of multiple tuples, e.g., in a join $R \bowtie S$ a tuple r that arrives at a S-store might produce $\{r \circ s_1, r \circ s_2, r \circ s_3\}$ if all satisfy the join predicate. This is emitted as a single Storm tuple containing a list of three data tuples.

Table 5.1 shows that the messages arriving at each instance of the same store



(a) Total number of tuples spread on all stores.

(b) Tuples placed in all stores over time.

(c) Distribution of stored tuples on tasks of the same store.

Figure 5.11: Storage occupation.

are evenly distributed, as well as the join results produced at each output. The plan which answers query MMM combines S_1, S_2, and S_3 in a MultiStream operator that materializes its results in store S_{123} and as root a MultiStream operator that joins that result with S_4. Due to the local join orders for the nested join, the stores for S_2 and S_3 receive more messages than S_1. This imbalance cannot easily be reduced by allocating more instances as this would only decrease the number of arriving store messages, not the probe messages, and it is part of ongoing work to find better solutions to this.

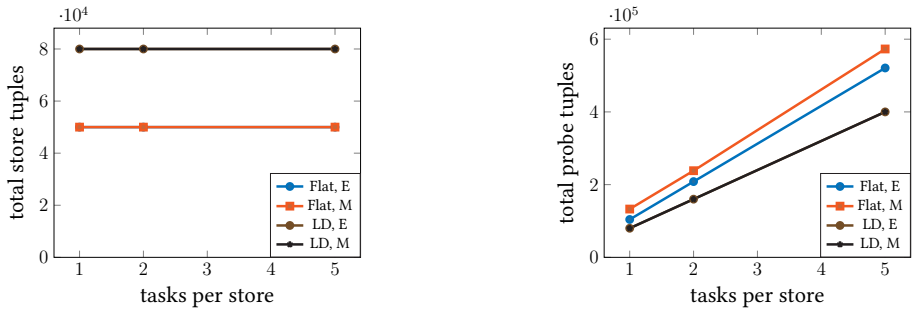
We also observe the amount of delayed joins. In this case, the arrival rate of the tuples is very high, thus tuples from S_0 are arriving earlier for probing at the S_{123}-store than the join results of $S_1 \bowtie S_2 \bowtie S_3$ which should be materialized. Hence, all joins that are conducted at the S_{123}-store are in fact delayed ones. This shows the importance of handling delayed joins in order to maintain correct results—and not only report a subset of the desired tuples.

5.5.7 Validity of Cost Models

We also validated the introduced cost models for required storage and caused communication. In order to do so, we measured sizes and rates offline for input data. Then we executed the queries and measured the according numbers inside the topology. Figure 5.12 shows this comparison for the extremes of a flat and a left-deep plan on the linear four-way query custom-*MMM* with relation sizes 10^4 and $M = 10^{-4}$. The comparison for estimated and measured storage cost is not visible, as the estimation and the amount of actually stored tuples is perfect. We notice a light overestimation of the probe cost for the query plans with longer probe orders. For the goal of deciding between different query plans, this is acceptable.

Store task	Input $[t/s]$	Output $[t/s]$	Delayed $[t/s]$
$S_{012,1}$	118.86	15.25	15.25
$S_{012,2}$	120.25	15.81	15.81
$S_{012,3}$	119.65	14.80	14.80
$S_{012,4}$	119.38	15.30	15.30
$S_{0,1}$	118.32	0.05	0.05
$S_{0,2}$	118.67	0.06	0.06
$S_{0,3}$	119.88	0.04	0.04
$S_{0,4}$	119.32	0.06	0.06
$S_{2,1}$	214.27	48.57	0.07
$S_{2,2}$	214.87	49.79	0.06
$S_{2,3}$	213.28	46.12	0.12
$S_{2,4}$	214.72	49.48	0.04
$S_{3,1}$	120.52	23.70	0.01
$S_{3,2}$	120.32	23.52	0.03
$S_{3,3}$	121.26	24.46	0.01
$S_{3,4}$	122.24	25.46	0.01

Table 5.1: Statistics on the average input messages per second, the produced join results per second, and the delayed messages per second for each store instance with four tasks per store.



(a) Storage cost

(b) Probe cost

Figure 5.12: Estimated cost (E) and measurements (M) from deployed plans with different degrees of parallelism.

Chapter 6

Windowed and Equality Joins

In this chapter, we add two aspects to the previous work that are commonly found in real-world streaming systems, namely *windows* and *equality predicates*.

Streaming queries operate by nature on unbounded input. This poses a problem we ignored in the last chapter, which is, the state required by a join operation eventually overflows the available memory. The common solution to this is to introduce *windows* [13, 18], periods of time in which relevant data lies, with the idea that data that is no longer in the current window does no longer contribute to the result and thus can be evicted from memory.

A consequence of this windowed view is that the interesting characteristics of the arriving and produced relations is no longer the size but the rate of the *streamed* relation. We begin this chapter with a discussion on query formulation and evaluation with windows and continue with the implications of the expiration and eviction of tuples.

Thereafter, we add understanding of equality predicates. If all, or a subset of, the queried predicates are equalities, we can exploit this in two ways: Equi-joins carry a straight-forward way of partitioning the operation across multiple compute nodes, thus they fit naturally into the computation model of partitioned scale-out systems like Flink or Spark. Locally, equi joins enable using indexes to speed up the match-making with stored items, reducing latency. While equi-join computation alone is not new, we tightly integrate it into our framework, allowing efficient computation of joins with a mixture of equalities and arbitrary predicates.

6.1 Windowed Join Queries

We extend queries by a notion of time-based sliding windows and write $R[W]$ when we want to express for some point in time t all tuples $R[W](t) := \{r \in R, r.\tau + W \geq t\}$.

The timestamp is a special attribute of a tuple, denoted $r.\tau$ for tuple r. Now, a join query over three relations R, S, and T can be written as $(R[W_R], S[W_S], T[W_T], \theta)$, where W_R, W_S, and W_T are the windows. If at some time t a tuple of R arrives, it can be joined with tuples from $S[W_S](t)$ and $T[W_T](t)$ which satisfy the join predicate.

Consider Figure 6.1, where a tuple of R arrived at time τ. According to the join predicate, this tuple finds three join partners in S, indicated by the blue lines. However, the window of S is evaluated for time τ and only contains the three tuples inside the box. Thus, the R-tuple is only joined with the two tuples indicated by the solid

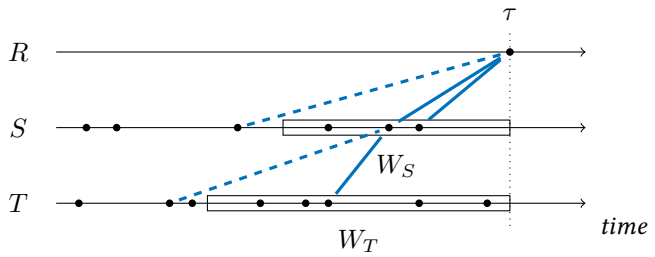


Figure 6.1: Join with two other relations restricted by windows.

line. Now, there is a partial result which needs to be joined with T. Again, the window of T is evaluated for time τ, thus only tuples inside the box drawn on T's time line qualify. Especially, for the second tuple the join predicate is satisfied (indicated by the line), but since it is outside the window-box, it must not be joined with the recently arrived R-tuple.

6.1.1 Sources of Time

In Section 5.4, we discussed the usage of timestamps in order to determine which store is responsible for producing a join result. This does not need to be an actual timestamp. In fact, it could be a random number, as long as it can be used to derive a strict order between all tuples. Now, the timestamp becomes a part of the correctness of the query result.

As discussed in Section 2.1, we differentiate between *event timestamps* and *system timestamps*. The event timestamp is a timestamp from the query domain, for example, if the input stream contains emails, it could be the date field of the header or the date of the last received field.

But there might not always be such a field in arriving tuples. In such cases, the system timestamp is assigned at arrival at the system. Inherently, this timestamp has an approximative character: Consider two (distributed) systems answering the same query with the same input tuples. In practice, the clocks on the underlying systems are diverging and thus, the system timestamp of the same tuple will differ in both systems. Consequently, the tuple with the earlier timestamp might find more join partners.

The benefit of system time, however, is that two tuples which arrive some timespan Δ apart, have a difference in timestamps of about Δ. With application time, as in the example with date headers of emails, tuples with timestamp difference Δ can actually arrive within a timespan of $\Delta' >> \Delta$, hence tuples need to be stored longer, and it becomes infeasible to guarantee correct join result for arbitrarily delayed tuples.

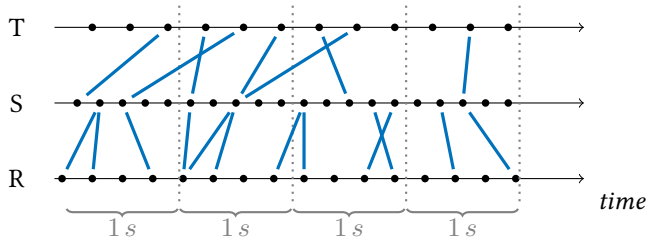


Figure 6.2: Continuous arrival of tuples with different rates.

6.2 Rate-based Optimization and Windows

When we restrict the relevant join partners by time windows, it does not make sense to measure sizes of relations. Instead, the content of a window is interesting. As the window duration is given by the author of the query, we need the **rate** of the streamed relation to get the content size. In this section, we explain how we migrate the size-based optimization from Chapter 5 to rate-based optimization, and thus can reuse algorithms.

Consider streamed relations R, S, and T which have tuples arriving with **rates** $|R|_\delta$, $|S|_\delta$, and $|T|_\delta$, such that δ is a time unit. It does not matter if δ is a second or a minute, and actually also each relation could be described using a different value for δ, but for the rest of this discussion we will use $\delta = 1\,s$. Figure 6.2 depicts how tuples of R, S, and T are arriving continuously with arriving rates $|R|_{1s} = 3\frac{1}{s}$, $|S|_{1s} = 5\frac{1}{s}$, and $|R|_{1s} = 3\frac{1}{s}$. If unambiguous, we will drop δ and the rate unit and from now on write, e.g., $|R| = 5$ to indicate that 5 tuples of R arrive per second.

The join **selectivity** is, analogously to its definition for static relations, the fraction of realized joins and possible joins; now per rate. This means, that if two relations have arrival rates of $|R|_{1s}$ and $|S|_{1s}$, and the join between has an output rate of $|R \bowtie S|_{1s}$, $f_{R,S}$ is $|R \bowtie S|_{1s}/|R \times S|_{1s}$.

Other parameters, like number of tasks and task capacity, however, remain static. For the number of tasks, nothing changes. But now we need to use rate times window size of a relation to check whether the task capacity with given degree of parallelism suffices to store the current prefix of that relation.

6.2.1 Windows and Intermediate Results

During tree construction we also implicitly introduce a materialized sub-query for each inner node. As it is materialized, it requires a window, otherwise we would not be able to estimate the required storage size. Let $R_1, \ldots, R_j$ be the tuples that are materialized and R_k be the relation that probes this materialized store.

Each of these relations R_i have a window W_i assigned. Then, r_k arrives and can

only join with tuples r_i with $r_i.\tau > r_k.\tau - W_i$. In the case of an intermediate result, this means for a each tuple $(r_1 \circ r_2 \circ \cdots \circ r_j)$ that $r_1.\tau > r_k.\tau - W_1 \wedge \cdots \wedge r_j.\tau > r_k.\tau - W_j$. This means, an intermediate relation needs to be annotated with the windows from all its inputs, and cannot have a single aggregated window.

6.3 Expiration and Eviction

With the introduction of windows, tuples *expire*, i.e., starting at some point in time they cannot be join partners of a new arriving tuple anymore, thus they do not contribute to results. Expired tuples can be *evicted* from the system, thereby freeing resources. But tuples are not necessarily evicted as soon as they expire.

6.3.1 Delayed Eviction

A tuple $r \in R[W]$ logically expires at time $r.\tau + W$, thus it could be evicted at $r.\tau + W$. Practically, however, tuples can arrive with a delay (see discussion about event time 2.1). Compared to operators that work on finished windows (like aggregation), a continuous join operator can just output delayed results and let downstream operators take care of it. This means, delayed tuples are handled by producing all join results with partners for this delayed tuple.

But allowing arbitrary delay means, not evicting tuples at all, which is not practical. Instead, users can define a eviction offset E, similar to Flink's maximum lateness or Spark's late threshold. This eviction offset is illustrated in Figure 6.3. Here the blue round dots indicate tuples arrived at R at some time in the past. They are placed on the time line according to their event-time timestamp.

Now at time τ a probe tuple s arrives, but its event-time timestamp is lower than τ. As discussed in Section 5.4, this tuple must not join with the r_3 to avoid producing the same output multiple times. Since s arrived with a delay smaller than E, when evaluating the window for this tuple, we need to be able to return $r_2 \circ s$, so r_2 must not have been evicted at this time.

Further, when tuple t arrives with a delay higher than E, there is a problem. As indicated by the red brace, t is in theory joinable with r_1. But since at the arrival of t, $r_1.\tau < \tau - W_R - E$ the system may have evicted r_1 already. At the same time, if $t \circ r_2$ are joinable, this result could be returned, so results could be incomplete. There are two options, (1) drop probe tuples with delay higher than E, and (2) allow them and return a result of unknown completeness. It is not clear, if such incomplete delayed results would harm the overall query, thus it should be configurable for the user of a system, how such tuples are handled.

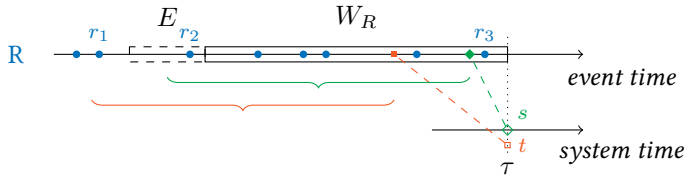


Figure 6.3: Window evaluations with delayed tuples.

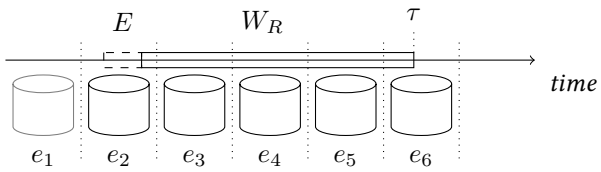


Figure 6.4: Sequence of epochs for R-store with their partitions.

For store tuples we have a similar problem, as very late tuple can still serve as join partner for another probe tuple. However, it might have missed tuples that were already evicted from the delayed probe buffer.

6.3.2 Lazy Eviction and Expiry Check

Inside the store tasks, we divide time into non-overlapping partitions. This is similar to Spark's DStreams [10] where data is located in time-sliced RDDs or BiStream's chained in-memory indexes [47]. Each partition, also called *epoch*, is responsible for a certain period of time. When an epoch is too old, i.e., its highest timestamp is smaller than $\tau - W - E$, its associated data can be evicted. When an epoch is young enough, i.e., its lowest timestamp is higher than $\tau - W$, tuples can be joined without even checking for timestamps.

Figure 6.4 shows epochs e_1 to e_6 and for time τ a window W as well as the eviction offset E. Arriving store tuples are placed into e_6 while the other epochs are stable (ignoring delay). When a probe tuple arrives at (event) time τ, it needs to evaluate the entire contents of e_3 to e_6, but for e_2 only tuples that satisfy $\theta \wedge r.\tau \geq \tau - W$. For e_1 we know that no tuple satisfies the second clause, thus it can be evicted.

6.4 Routing and Partitioning

In Chapter 5, we built a routing strategy that ensures that every pair of tuples can see each other. We now restrict the placement of tuples at stores and explain how to adjust the routing strategy.

6.4.1 The MultiStream Operator for Equi-Joins

The high-level idea is this: instead of partitioning each relation randomly over the tasks of a store, partition it according to an attribute of an equality predicate, and then use this to set up a routing strategy that avoids broadcast messages where possible.

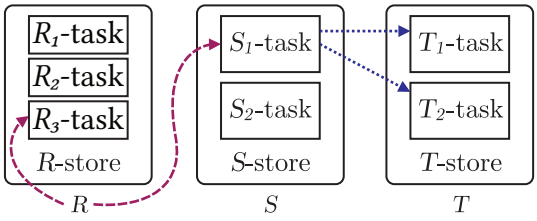
Consider a query $(R, S, T, R.a = S.a \wedge \theta_{S,T})$ that we compute using a MultiStream operator with these adjustments for equi-joins. Tuples of T are distributed randomly on the T-store as before. Tuples of R and S are now partitioned according to their a attribute. In Figure 6.5 we show a part of the routing for this operator. When a tuple of R arrives, it obviously knows the value of $R.a$, thus it is sent directly to the task assigned to that value. As join results need to satisfy $R.a = S.a$, this tuple can be sent directly to the S-task containing S-tuples with the same a value. After that, tuples need to be broadcast to all T-tasks.

Figure 6.5b shows the routing for a T-tuple, which starts like in the theta case: The tuple is sent to a randomly chosen task of the T-store, and is then broadcast to both S-tasks. There, intermediate results are computed, and now they are sent to the R-store. In the figure, connections are drawn between all tasks, but in this case they are no broadcasts. Figure 6.5c illustrates this for a tuple $(5) \in T$, where we assume for simplicity that $\theta_{T,S} = true$ and S and R only consist of their a attribute. We see that (5) is sent to both tasks of the S-store where each stored tuple is joined with the arriving one. The example is constructed such that all matching join partners for the output of a S-task would be found on different R-tasks, and thus, each R-task receives messages. However, in contrast to the theta-join computation, these messages do not contain duplicates, so the number of tuples sent is here independent from the degree of parallelism of the receiving store.

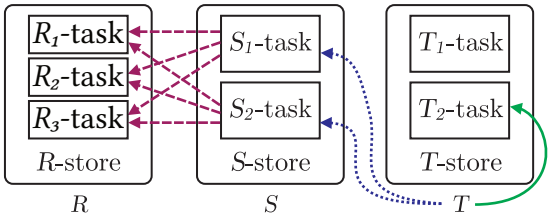
Multiple Equi-Joins

In the previous example, θ contained only one equality clause, thus it was straightforward to partition the stores. When more or even all predicates are equalities, we need to choose the partitioning for some of the stores. Consider now query $(R, S, T, R.a = S.b, S.c = T.d)$, where there are two options.

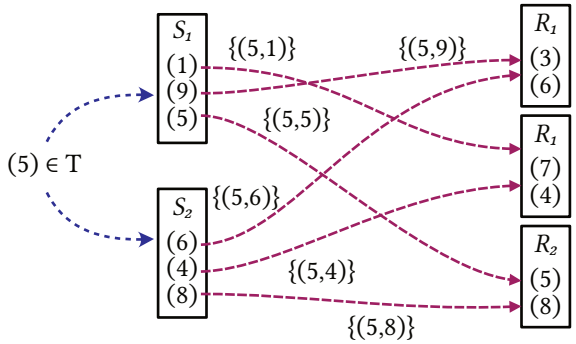
Option 1: partition S according to b. Then tuples of R can be sent to the S-task indicated by their a-value. All resulting tuples ($\subseteq R \bowtie S$) do have a c-attribute and can then be sent to the correct T-task. Tuples $t \in T$, however, do not have an attribute with a value that can indicate the S partition. This means, tuples of S with a c-value of $t.d$ can be found in every partition of the S-store. Thus, t has to be broadcast to all partitions of the S-store. The result of this join ($\subseteq S \bowtie T$) consists of tuples with a b-attribute which can be used to look up the according R-partition.



(a) Routing for tuples of R.



(b) Routing for tuples of T.



(c) Concrete example for the path of a tuple of T.

Figure 6.5: MultiStream operator where R and S are partitioned for $R.a = S.a$.

Option 2: S is partitioned according to c. Then tuples of T can be directly sent to the correct partition of S without broadcasting. But now R lacks this information, as a can no longer indicate the correct partition, so R tuples have to be broadcast to the S-store. In both cases, tuples from S can be sent without broadcasting either first to R and intermediate results to T or the other way round.

The Adjustment of MultiStream

The previous examples showed, how we can configure a MultiStream operator in presence of equality clauses. In general, this is a **partitioning scheme** $\mathcal{P}$, i.e., a

complete mapping from stores to attributes: $\{R_1, \dots R_n\} \mapsto \mathcal{A}^2$ where mapping to the empty set indicates random partitioning; and adjusted probe orders, where in each step it is known which attribute to use for addressing the receiving task of the next step.

6.4.2 Optimizing Plans

First, we need to update the cost formula to include partitioned stores. The storage cost remains the same, as the amount of tuples to be stored does not change.

Given a partitioning scheme $\mathcal{P}$ and the probe orders $\sigma_1, \dots, \sigma_n$. Then the total cost for probing expressed in number of tuples is:

$$\text{PCost}(O) = \sum_{1 \le i \le n} \sum_{2 \le j \le n} \chi_{\sigma_i(j)} \cdot \prod_{k=1}^{j-1} W_{\sigma_i(k)} \cdot \prod_{k=1,k'=1}^{k=j-1,k'=j=1} \theta_{k,k'} \cdot \frac{1}{j}, \tag{6.1}$$

with

$$\chi_{\sigma_i(j)} = \begin{cases} 1 & \text{if } \mathcal{P}(R_{\sigma_i(j)}) \subseteq \bigcup_{1 \le k \le j} \mathcal{A}(R_{\sigma_i(k)}) \\ N_i(j) & \text{otherwise.} \end{cases} \tag{6.2}$$

Informally, χ is used to quantify the number of copies that are simultaneously sent during probing of the j-th store when a tuple from relation R_i arrives. If all attributes which the $R_{\sigma_i(2)}$ store is partitioned after are known, the probe tuple can be sent directly to the desired partition. However, if only one attribute is missing, this is not possible anymore, and it has to be sent to each partition, which is reflected by $N_i(j)$.

6.4.3 Optimization

The search space for optimization now includes another dimension, which is the partitioning of stores. We can use the query to determine a set of partitioning candidates for each (potential) store. For each input relation the candidate attributes are these attributes that are referenced in an equality predicate, e.g., if $R.a = S.b \in \theta$, then a is a candidate attribute for R and b is a candidate attribute for S. For a materialized intermediate result, all attributes that are referenced by equality predicates between a relation of this intermediate result and another relation are candidates.

What this means can easier be understood by looking at the query graph in Figure 6.6. This graph represents a query over five inputs that should be joined using the equality predicates annotated at the edges. Now we want to find partitioning candidates for the store materializing $(R_2, R_3, R_4, R_3.e = R_4.f, R_2.g = R_4.h)$, as marked by the blue area. The predicates that can contribute to the partitioning candidates are the ones leaving the blue area, and the attributes are the ones referencing a relation inside the blue area, marked yellow. Thus, we can use $R_2.b$, $R_3.d$, or $R_4.i$ for parti-

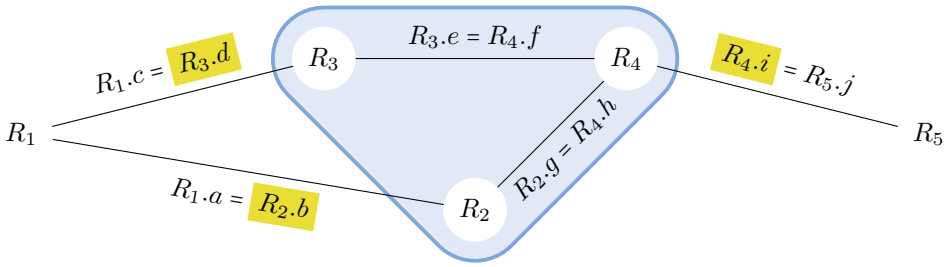


Figure 6.6: Query graph for a query over five relations with predicates annoteted at the edges. The blue area marks a potentially materialized sub-relation.

input: query $q = (R_1, \dots R_n, \theta)$
1 $C \leftarrow \{\}$
2 **for** $(R_i.a = R_j.b) \in \theta$
3 $\quad C[R_i] << a; C[R_j] << b$
4 leastCost = ∞, bestPlan = $\perp$
5 **for** $\mathcal{P} \in$ `createPartitioning`(C)
6 $\quad$ (cost, plan) = `optimize`$(q, \mathcal{P}, \cdot)$
7 $\quad$ **if** (cost < leastCost) = bestPlan $\leftarrow$ plan
8 **return** plan

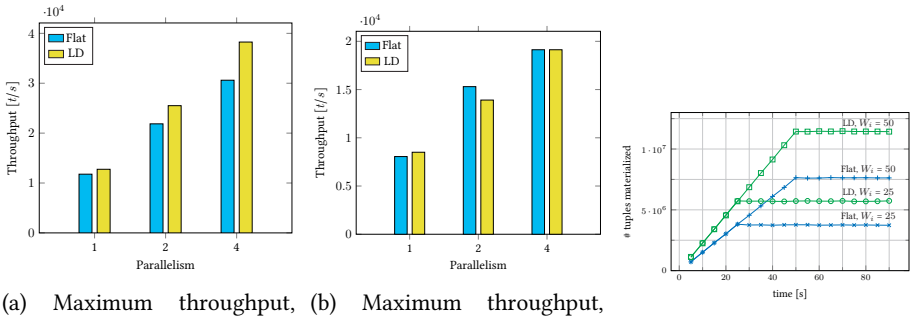
Figure 6.7: Enumerating partitioning strategies.

tioning. These attributes are all known when a result is sent to the R_{234}-store, but for example $R_3.e$ is also known. However, it makes no sense to use $R_3.e$ for partitioning, as it will no longer be used to find join partners.

Enumerating Partitioning Candidates

Let a query have at most one predicate $R_i.a = R_j.a'$ between any two relations R_i, R_j referencing different attributes a and a' each. Then the number of possible partitioning schemes $\mathcal{P}$ is the product of the degrees of all relation nodes in the query graph. In the worst case there is a predicate for each relation. Then there are $n \cdot (n-1)$ partitioning schemes for n relations. If we allow multiple predicates between the same pair of relations, e.g., $R_i.a = R_j.a'$ and $R_i.b = R_j.b'$, this can be interpreted as allowing multiple edges between nodes in the query graph, thus the worst case is not bounded by the number of relations anymore, but by the predicates. The best case, is for all predicates that reference relation R_i to be of form $R_i.a = R_j.b$ with some fixed a. Then, R_i's only choice is to be partitioned according to a.

With these considerations, we enumerate the candidates as shown in Figure 6.7.



(a) Maximum throughput, window size 25s.

(b) Maximum throughput, window size 50s.

(c) Materialization of tuples.

Figure 6.8: Throughput and materialization of Q3 using different window sizes.

6.5 Experiments

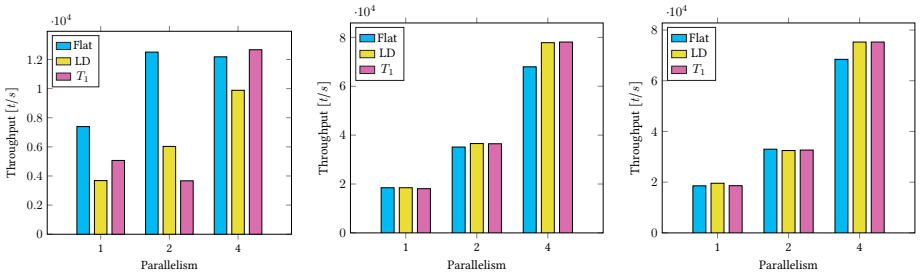
We continue the experimental evaluation of Section 5.5 and extend the observations to windowed joins and equality predicates. For the computation of windows we use system time by applying the current time at the spout that reads a tuple into the topology.

6.5.1 Windowed Throughput

For examining the effect of window sizes on throughput we cannot just feed tuples as fast as possible into the topology. The first tuples would first be processed and later tuples only if enough processing resources are free, thus later tuples would be delayed, and thus contents of the windows would vary too much. Instead, we start with a low input rate and increase the rate to find the lowest rate such that Storm does not need to apply backpressure. We then report on the throughput based on this rate. This strategy of computing throughput is also called the maximum sustainable throughput [58].

In Figure 6.8, we show the results for execution of Q3 with theta-predicates and window sizes of 25 and 50 seconds. First, 6.8a shows the throughput for varying degrees of parallelism using global scaling with a window of 25 seconds. As Q3 spans three input relations, the flat and the left-deep tree are all possibilities. We see that both plans are scaling sublinearly, which is presumably due to the communication overhead of the theta-join computation. The left-deep plan's relative performance gets better with higher task parallelism, as the additional intermediate result saves probing.

Figure 6.8b shows a similar behavior of the plans, this time for a 50 second window. The trend is the same, however the achievable throughput varied more. With parallelism of 2, the flat tree was able to achieve a higher throughput and the achievable



(a) The theta variant, $Q2_\theta$. (b) The equi variant, $Q2_=$ (c) The mixed variant, $Q2_{\theta=}$.

Figure 6.9: Throughput and parallelism of different variants of Q2.

rate was actually the same. In general, the throughput was about half of the throughput with 25 second windows, which we attribute to the lower amount of nested-loops joins.

This can be better understood by looking at Figure 6.8c, where we see the number of materialized tuples over time. Every five seconds, we emit a marker tuple into the topology which forces workers to log the current number of materialized tuples. The sum of these logged numbers is shown in the graph. At the beginning of the computation, the number of materialized tuples grows until the first window duration is reached. After this had happened, the sizes stay the same, and we see that the plans executing a twice as big window also contain twice as much tuples. Further, as seen before, the left-deep variant stores significantly more tuples than the flat plan.

6.5.2 Equi-Join Throughput

We looked at the throughput of Q2 executed in three variants. $Q2_\theta$ is the variant like described in Section 5.5, where each predicate is handled as theta-join predicate. Then we let CLASH handle all predicates as equalities in $Q2_=$, thus each store is partitioned and executes hash joins locally. Finally, as combination of both, $Q2_{\theta=}$ handles all but the predicate between the two biggest inputs, `part` and `partsupp`, as θ joins. The results in Figure 6.9 show the throughput for a flat plan consisting only of a single MultiStream operator joining all five inputs, the optimal left-deep tree (LD), and T_1 is again the optimized tree by CLASH. Figure 6.9a is a repetition of Figure 5.8a with the throughput results of running Q2 in pure theta mode. Figure 6.9b shows the pure equality mode, where each tuple has a single destination and locally no nested-loops joins are used. In this figure, all three trees scale linearly, altough the flat tree is a little shy of the other two, presumable because it still has more probe work to do.

It is remarkable, that the variants with equi joins seem to be much more stable. We attribute this to stronger variances in performance of the EC2 machines that host these experiments. Also, due to the reduced work, the achieved throughput is an order

of magnitude higher than with theta joins.

Finally, when we examine Figure 6.9c, we see, that this variant nearly has the same throughput as the results for $Q2_{=}$. This can be explained by the rather small fraction of theta joins which negatively impact the performance.

Chapter 7

Multi-Query Optimization and Adaptive Join Processing

In this chapter, we extend our approach to answering multiple queries at once and introduce ways to adaptively change the processing strategy when query workload or underlying data characteristics change. Due to the long standing nature of streaming queries, it is almost natural to try to share computation paths or intermediate results between multiple queries.

Previously, the main focus was on tree-based optimization where a tree of relations is built. Now, we shift the focus to the probe orders and see materialization points merely as artifact of the choice of probe orders.

7.1 Optimization using Integer Linear Programming

An integer linear program (ILP) is in general an optimization problem that determines assignments for a set of variables such that a cost term is minimized (or maximized) [54]. The cost term is the inner product of the user-defined cost for each variable c_i and the integer variable x_i, so $c_1x_1 + c_2x_2 + \dots$ is subject to minimization (or maximization). Further, these variables have to fulfill a set of constraints, all given in the form of $a_{1,1}x_1 + a_{1,2}x_2 + \dots \geq b_1$.

We follow the approach of [23] for formulating a multi-query optimization problem as ILP. Consider a query q_i for which we have alternative join plans $p_{i,1}, \dots, p_{i,k}$ to choose from. For each such query q_i, we generate equations for the ILP:

$$x_{i,1} + x_{i,2} + \dots + x_{i,k} = 1 \tag{7.1}$$

where $x_{i,j} = 1$ iff plan j is chosen for query i. As the variables $x_{i,j}$ are integers, these equations are satisfied iff exactly one plan is chosen for each query. Each plan is composed of multiple tasks which represent the computation of a subresult and have cost assigned. For example, plan $p_{i,1}$ is composed of tasks $t_1, \dots, t_r$ with cost $c_1, \dots, c_r$ respectively. Then, we also add equations

$$-Cx_{i,1} + c_1x_{t_1} + \dots + c_rx_{t_r} \geq 0 \tag{7.2}$$

where $C := \sum_{i=1\dots r} c_r$. Thus, if plan $p_{i,j}$ is chosen, $x_{i,j}$ is set to 1 and negative cost have to be balanced by selecting all the associated tasks. The same tasks might appear

Algorithm 4 Candidate probe order construction algorithm.

```
input:  query q, MIR
output: candidate probe orders
1  fun construct_rec(head)
2     result ← [ ]
3     for r ∈ joinable(q, head, MIR)
4        newHead ← head + r
5        if newHead is complete
6           result ← result + [newHead]
7        else
8           result ← result + construct_rec[newHead]
9
10 for relation in query
11    construct_rec(relation)
```

in candidate plans for different queries, and thus, if such plans are selected, computation can be shared between these plans. In total, the sum of costs times tasks is subject to minimization. Now we need to generate candidate plans (in this case probe orders) to chose from, and then translate these choices into topologies.

In order to translate the query set into an ILP, we first create for each query *materializable intermediate results* (MIR) and, based on that, a set of candidate probe orders. An MIR consists of a subset of the queried relations and the join predicates defined on them such that cross products are avoided. For example, for query $(R, S, T, R.a = S.a, S.b = T.b)$ the materializable intermediate results would be $(R, S, R.a = S.a)$ and $(S, T, S.b = T.b)$ but not (R, T). An MIR can be seen as the relation computed by a potential inner MultiStream operator when constructing operator trees.

The candidate probe orders are determined using Algorithm 4. For each relation in the query the recursive sub-function `construct_rec` is called in order to construct probe orders from head to tail. It returns all probe orders that can be used to answer q if the starting tuple is the result of joining `head`. In this sub-function in Line 3, we iterate over all MIRs which are, according to the given query, joinable with the current head. This way, we avoid producing cross products. If joining head and r yields a complete result, i.e., all input relations of q are covered, the probe order is completed. Otherwise, the same function is recursively called to yield all probe orders that start with the previous head joined with r. We assume here that there are no queries which include a cross product. For this case one can revert to constructing probe orders as described in Chapter 5 by adding artificial `true`-join predicates.

We further need candidate attributes for partitioning of the MIR stores. For an $r \in MIR$ these are all attributes which define a join with another relation that is not part of r. To give an example, if for query $q = (R, S, T, R.a = S.a, S.b = T.b)$ the intermediate

Algorithm 5 ILP construction procedure.

input: queries Q, probe order candidates C,
partitioning candidates P
output: ilp constraints A, optimization goal G

```
1 A ← {}
2 for q ∈ Q, r ∈ rel(q)
3   p ← apply_partitioning(C[r], P)
4   for σ ∈ p
5     A ← A ∪ cost_constraint(σ)
6   A ← A ∪ probe_order_constraint(p)
7 G ← goal(A)
```

result $(R, S, R.a = S.a)$ is materialized, a is *not* a candidate for partitioning, because there is no join with T that uses this attribute. However, attribute b is a candidate for partitioning. This makes sense because all tuples that are sent to the RS-store know the value of the b-attribute and, hence, can be routed correctly. Also, partitioning according to a implies that tuples from T need to be broadcast to all T-tasks, while with a partitioning according to b this full broadcast is avoided.

While the input relations are always materialized, this is not necessarily the case for intermediate relations. They are only required if a probe order is selected which also uses those relations. Thus, for intermediate relations, we also generate probe orders using the subquery for the intermediate result as input to the candidate probe order construction.

Based on probe order and partitioning candidates, we construct the ILP as shown in Algorithm 5. In variable A, we collect all constraints that the ILP must satisfy. In Line 2, we iterate over all combinations of queries and possible starting relations to add constraints that select a probe order for each starting relation. Therefore, the partitioning is applied to the probe order candidates for the starting relation, $C[r]$. In Line 3, variable p contains probe orders where all MIRs are decorated with the partitioning attribute. This is necessary for building the cost constraint (Line 5). The cost values are set according to Equation 6.1 and in order to compute χ, it is necessary to distinguish between differently partitioned stores.

For computing $probe\_order\_constraint(p)$, with probe orders $\sigma_1, \ldots, \sigma_n \in p$, for each probe order σ_i a new variable $x_i \in \{0, 1\}$ is introduced.

$$x_1 + x_2 + \cdots + x_n = 1$$

This line resembles Equation 7.1. If the probe order identified by x_i contains a materi-

alized intermediate result over relations $1, \ldots, l$, this also has to be computed. Hence for each of the inputs, a probe order which creates the intermediate result needs to be installed, which is made sure by the following constraints:

$$
\begin{aligned}
&- k_1 \cdot x_i + x'_{1,1} + x'_{1,2} + \cdots + x'_{1,k_1} \geq 0 \\
&\ldots \\
&- k_l \cdot x_i + x'_{l,1} + x'_{l,2} + \cdots + x'_{l,k_l} \geq 0
\end{aligned}
$$

Here, k_j is set to the number of probe orders required for computing the result starting from relation $j \in 1, \ldots, l$. Variables x' indicate if the probe order for that subquery will be executed. Since each line needs to be non-negative, it is guaranteed that the intermediate result is actually computed.

The $cost\_constraint(\sigma)$, which we will model using Equation 7.2, is composed of the cost of all the prefixes of that probe order. With $\sigma = \langle S_1, S_2, \ldots, S_m \rangle$ these prefixes are $\sigma_1 = \langle S_1, S_2 \rangle$, $\sigma_2 = \langle S_1, S_2, S_3 \rangle$ up to m. For each prefix we introduce a **step variable** y_i, and it is crucial, that all equal prefixes used in candidates of other queries get the same variable y_i assigned. For each of these prefixes, we also introduce the **step cost** which is the innermost term of the inner sum in Equation 6.1. For a probe-order prefix $\langle S_1, S_2, S_3 \rangle$ this is the cost of sending the partial result $S_1 \bowtie S_2$ to S_3. Thus, for each probe order σ the following constraint is added:

$$
\begin{aligned}
-PCost(\sigma) \cdot x_1 + StepCost(\sigma_1) \cdot y_1 \\
+ \cdots + StepCost(\sigma_m) \cdot y_m \geq 0
\end{aligned}
$$

In Line 7, the goal is set. The goal is derived from the step cost and step variables of the previously added constraints:

$$
\min \sum_{i=1 \ldots m} StepCost(\sigma_i) \cdot y_i
$$

As $y_i \in \{0, 1\}$, the value of the sum is only affected by the variables set to 1. Only combinations of variables y_i can be set to 1 such that all queries have all necessary probe orders for computing their results. Thus, a solution that minimizes this term can also be translated to a correctly working topology. This topology needs to be deployed to a stream processor like Apache Storm and processes the query.

ILP Creation Example

Consider the two three-way queries $q_1 = (R, S, T, R.b = S.b, S.c = T.c)$ and $q_2 = (S, T, U, S.c = T.c, T.d = U.d)$. In Figure 7.1, we see first the materializable intermediate results composed of the input relations as well as the intermediate results. Here, RS stands for the result of the subquery $q_{RS} = (R, S, R.b = S.b)$. Since we have potential intermediate results, they also need to be created, and thus, probe orders for them have to be installed as well. The probe orders are listed next. There is one probe order per input relation of each query. For example, q_1 consists of three inputs and hence, three sets of probe orders are created and one of the candidates of each set needs to be used.

Thereafter, the partitioning is applied to the probe orders. In Figure 7.1, we only show the options for probe orders for q_1 and R. Here it is interesting to see, that also partitioning which is not beneficial to the current query is included. For example, the probe order $\langle R, S[b], T[d] \rangle$ indicates that the S-store is partitioned according to attribute b and the T-store is partitioned according to d. If this probe order is installed, a tuple from R, after it probed the S-store, needs to be broadcast to all T-workers in order to compute the result for q_1, because this tuple does not contain the value of attribute d. The partitioning of T according to d is only useful for q_2.

Finally, the constraints for the ILP are added. The first constraint requires that exactly one from the probe order candidates σ_1 to σ_6 is chosen. For this we add an ILP variable x_i for each σ_i that takes values in $\{0, 1\}$. Then, we need to make sure, that for probe orders which include intermediate results, these intermediate results are actually computed. The next constraint showcases this for σ_5. In this probe order, R-tuples are sent to the ST-store which is partitioned according to b for probing. To do so, the ST-store needs to be installed and also kept up to date with this intermediate result. In turn, probe orders for computing $S \bowtie T$ need to be installed. In this case, there are four probe orders, one for sending S to the T-store and one for sending T to the S-store and each store can be partitioned according to two attributes. Out of these probe orders we need two (one for each relation) and thus we add constraints 2 and 3.

Actually, the computation of the intermediate result is independent from the partitioning of this result's store. Thus, the same intermediate result computation can be used for σ_6. We then need to make sure that each probe order, if it is chosen, is computed correctly. Probe order σ_1, for example, has the prefix σ_7. Thus we add constraint 4 where ILP variables for each step in that probe order are set, y_7 and y_1. These variables are associated with the step cost for σ_7, which is the cost for sending tuples from R to the S-store that is partitioned by b, and the step cost for σ_1, which is the cost for sending tuples from the S-store to the T-store that is partitioned by c. In constraint 5, the next probe has the same first step, and thus, it is crucial that the

same variable y_7 is put into the ILP, otherwise the ILP would not take the shared work for y_7 into account. The optimization goal of the ILP is then to minimize the sum of the step costs of all used steps.

$q_1 = R(b), S(b,c), T(c)$, $q_2 = S(c), T(c,d), U(d)$
$MIR = R, S, T, U, RS, ST, TU$
Candidate probe orders:
for q_1:

- **R**: $\langle R, S, T\rangle$, $\langle R, ST\rangle$
- **S**: $\langle S, T, R\rangle$, $\langle S, R, T\rangle$
- **T**: $\langle T, S, R\rangle$, $\langle T, RS\rangle$

for q_2:

- **S**: $\langle S, T, U\rangle$, $\langle S, TU\rangle$
- **T**: $\langle T, U, S\rangle$, $\langle T, S, U\rangle$
- **U**: $\langle U, T, S\rangle$, $\langle U, ST\rangle$

for q_{RS}:

- **R**: $\langle R, S\rangle$
- **S**: $\langle S, R\rangle$

for q_{ST}:

- **S**: $\langle S, T\rangle$
- **T**: $\langle T, S\rangle$

for q_{TU}:

- **T**: $\langle T, U\rangle$
- **U**: $\langle U, T\rangle$

Probe orders with partitioning for q_1 and R, including probe order prefixes:

σ_1 $\langle R, S[b], T[c]\rangle$
σ_2 $\langle R, S[c], T[c]\rangle$
σ_3 $\langle R, S[b], T[d]\rangle$
σ_4 $\langle R, S[c], T[d]\rangle$
σ_5 $\langle R, ST[b]\rangle$
σ_6 $\langle R, ST[d]\rangle$
σ_7 $\langle R, S[b]\rangle$
σ_8 $\langle R, S[c]\rangle$

Constraints:

1. $x_{\sigma_1} + x_{\sigma_2} + x_{\sigma_3} + x_{\sigma_4} + x_{\sigma_5} + x_{\sigma_6} = 1$ (one probe order)
2. $-2x_{\sigma_5} + x_{\sigma'_1} + x_{\sigma'_2} \geq 0$
3. $-2x_{\sigma_5} + x_{\sigma'_3} + x_{\sigma'_4} \geq 0$ (subqueries for σ_9)
4. $-PCost(\sigma_1) \cdot x_{\sigma_1} + StepCost(\sigma_7) \cdot y_7$
   $+StepCost(\sigma_1) \cdot y_1 \geq 0$
5. $-PCost(\sigma_3) \cdot x_{\sigma_3} + StepCost(\sigma_7) \cdot y_7$
   $+StepCost(\sigma_3) \cdot y_3 \geq 0$

   ...

Optimization goal:

$$\begin{aligned}\min \quad & StepCost(\sigma_1)y_{\sigma_1} + StepCost(\sigma_3)y_{\sigma_3} \\ & + StepCost(\sigma_7)y_{\sigma_7} + \ldots\end{aligned}$$

Figure 7.1: Deriving an ILP for queries q_1 and q_2.

Multi-Query Optimization Example

In this example, we only focus on choosing probe orders and ignore materializing subqueries and partitioning. Thus, we ignore additional cost for broadcast and do not write the partitioning attribute. Consider the queries $q_1 = (R, S, T, R.a = S.a, S.b = T.b)$ and $q_2 = (S, T, U, S.b = T.b, T.c = U.c)$ where each relation streams at a rate of 100 tuples per time unit and the join between S and T produces 150 intermediate results, while the other join produces only 100 intermediate results. We now focus on what happens with relations S in q_1 and T in q_2. Optimizing each query individually, we would install the probe orders $\langle S, R, T\rangle$ and $\langle T, U, S\rangle$ in order to avoid the more expensive intermediate join between S and T, and send in total 475 tuples for probing in each query, thus 950 tuples in total. Since for answering q_1 (respectively, q_2) correctly tuples must to be sent from T to S (S to T), we can exploit this and instead install probe order $\langle T, S, U\rangle$ ($\langle S, T, R\rangle$) for q_2 (q_1).

For the optimization problem we assign variables for the steps in the probe order, e.g., x_{RS} for the cost of sending R-tuples to the S-store for probing, or x_{RST} for the cost of sending the intermediate result of $R \bowtie S$ to the T-store for probing. The cost associated with these variables is 100 for all first steps, and 75 for joins between S and T and 50 for the other joins (c.f. Formula 6.1). For q_1 and starting relation S the following constraint rows are added to the ILP:

$$\begin{aligned} x_1 + x_2 &= 1 \\ -150x_1 + 100x_{SR} + 50x_{SRT} &\geq 0 \\ -175x_2 + 100x_{ST} + 75x_{STR} &\geq 0 \end{aligned}$$

x_1 stands for the probe order $\langle S, R, T\rangle$ and x_2 for the probe order $\langle S, T, R\rangle$. The first line makes sure that only one of these variables can be 1 and this variable determines which of the probe orders will be installed in the running topology. The second line enforces that if x_1 is set to 1, then also x_{SR} and x_{SRT} are set to 1.

For q_2 and starting relation S there is only one probe order, thus the following constraints rows are added to the ILP:

$$\begin{aligned} x_3 &= 1 \\ -150x_3 + 100x_{ST} + 75x_{STU} &\geq 0 \end{aligned}$$

Essentially, this leaves no choice: x_3 has to be set, and consequently also x_{ST} and x_{STU} have to be set, and thus S tuples need to be sent to T and afterwards to U in

order to produce all desired join results. The optimization goal then includes the here mentioned cost variables and more which are not shown for clarity:

$$\min \quad 100x_{SR} + 50x_{SRT} + 100x_{ST} + 75x_{STR} + 75x_{STU}$$

As discussed, x_{ST} and x_{STU} need to be set to 1 due to q_2. This way, selecting the probe order x_2 (and thus setting x_{STR} to 1) adds only 75 to the cost. Selecting x_1, on the other hand, requires $x_{SR} = x_{SRT} = 1$ and adds 150 to the cost. Hence, the locally—for query q_1 in isolation—suboptimal probe order x_2 is chosen and an overall lower number of tuples needs to be sent around.

7.1.1 Analysis

The number of materializable intermediate results of a query over n relations is in the worst case 2^n when the query graph is a clique, i.e., for every pair of relations there is a join predicate. For example, for a linear query, the size of MIRs is the number of consecutive subsequences of a word of length n, so only $n(n+1)$. The number of candidate probe orders per query and relation is, in the worst case, the number of permutations of these subsequences times the number of partitioning options. This all heavily depends on the query. For example, for a linear query there are 2^{n-2} and a star query has only $n-1$ partitions to choose from. The number of ILP variables is then for all queries the sum of the amount of candidates for each query, as well as the prefixes of the probe orders.

7.1.2 Transformation to Executable Strategies

The result of the ILP optimization is the assignment of probe order variables (and step variables, but we can ignore them). We now detail on how to construct a topology of compute tasks for actually computing the query.

The probe orders with variables set to 1 are the probe orders that should be used in the actual query execution. We merge probe orders into probe trees, as illustrated in Figure 7.2. Here, we see several probe orders for the starting relation R. Since σ_1 and σ_2 both have the same first step, probing the S-store, they are represented by the edge from R to the node with label $S[d]$. Multiple outgoing edges in this graph indicate that a tuple is copied and sent to both targeted stores. This is done for all probe orders, such that we end up with a forest of such probe trees. For each distinct label of the inner nodes, a store is introduced in the topology. This way, nodes with the same label in different probe trees refer to the same store and data is not stored redundantly. For the roots, ingestion methods (in case of Storm these are Spouts) need to be installed. For each edge of a probe graph, a new, unique, edge label is introduced.

Algorithm 6 Non-adaptive version of incoming tuple handling procedure.

```
1  fun handle(e_in, tuple)
2    rules ← ruleset[e_in]
3    for rule in rules
4      switch type(rule)
4        case StoreRule: store(tuple)
5        case ProbeRule: probe(tuple)
```

With help of these edge labels, rules are registered at all stores. These rules define the behavior of the store for a received tuple based on the incoming edge label. The sending store is not enough, as there might be tuples from different probe trees sent from one to the other store. These tuples stem from different (sub)relations and the probe result is sent to different stores for further processing, so we use the edge labels instead. A rule follows the pattern *if tuple arrives from edge E_{in}, probe using predicate P, and send result (if any) to E_{out}*. All rules registered to a store are organized in a ruleset. On each arriving tuple this ruleset is consulted for deciding how to proceed with the tuple. During runtime, Algorithm 6 is used to decide on a worker how to process a tuple: in Line 2, the matching rules for the incoming edge are extracted. Since this is done for every tuple, this must happen quickly, so the ruleset is organized as hash map keyed by the incoming edge labels. Then the type of the rule decides how the arriving tuple has to be handled. If the rule is a *store rule*, like in Line 4, the arriving tuple is added to the local store of arrived tuples, and is ready for other later arriving probe tuples. These arrive over edges where a *probe rule* is registered. If such a tuple arrives, Line 5 makes sure it probes with the previously arrived tuples of the stored relation.

A probe rule contains a description of the way of accessing the tuples. For example, a tuple sent via s_3 in Figure 7.2 contains a partial result of $R \bowtie S$, and the T-store contains the previously arrived tuples of T. For the local probe handling at workers it is irrelevant how the store is partitioned. Consider here that the probe should determine join partners for the predicate $R.b = T.c$. The probe rule accesses the $R.b$-attribute of the incoming tuple and needs to find all stored tuples with the same value in $T.c$ for creating join results. For each distinct attribute access in a store, indices are created locally for efficiently answering probe request.

We implement this ILP-based optimization in CLASH as individual optimizer that produces a physical graph. With this physical graph, the same topology generation and execution infrastructure as in our previous approaches can be used.

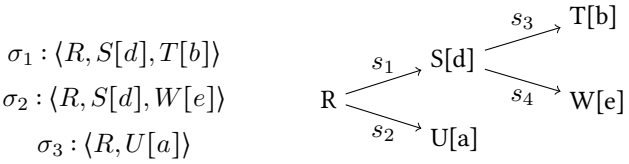


Figure 7.2: Three probe orders for the same starting relation merged into a probe tree.

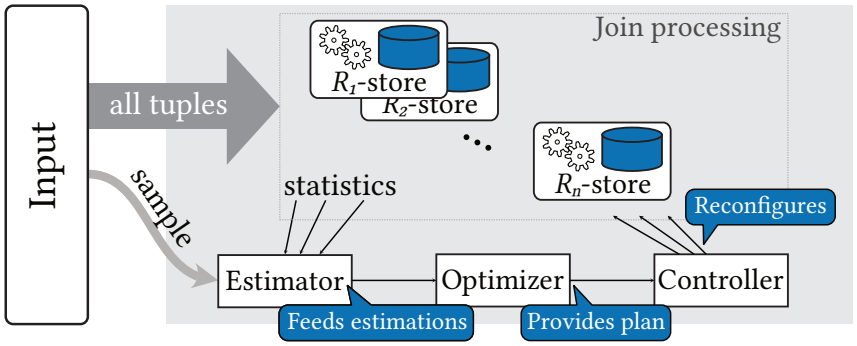


Figure 7.3: Extension of our join processing architecture for estimator, optimizer, and topology controller.

7.2 Continuous Optimization

In this section, we detail on the process used for continuously deciding on optimizing the query plan in use. The high-level aspects can be seen in Figure 7.3, where the dotted box in the middle houses the join processing stores as explained in the previous chapters. Before, optimization was done offline, i.e., the query and estimation data was sent to the optimizer, the optimizer produced a plan and consecutively CLASH built a topology which is submitted to a Storm cluster. Now, optimization becomes an online process. For this, first estimations need to be continuously updated. The estimator component periodically receives statistics from the stores, as well as a sample from the input. Based on this, the estimator provides estimations to the optimizer. The optimizer does the same as before, and provides a query plan. This plan is now sent to the controller, which is able to instruct the stores to behave differently, e.g., by installing new probe orders.

7.2.1 Estimation

For estimation, we use a combination of sampling and result interpretation. The idea is, that we can exactly measure the performance of the installed probe orders. For potentially better probe orders, we can only make educated guesses based on statistics.

Consider the following example. We have a three-way join query $(R, S, T, R.a = S.a, S.b = T.b)$, and currently probe order $\sigma_S = \langle S, R, T \rangle$ is in place. From this probe order, we can actually observe selectivities $f_{R,S}$ and $f_{R \bowtie S,T}$. However, there is another valid probe order $\sigma'_S = \langle S, T, R \rangle$. In order to estimate the performance of σ'_S we need to estimate $f_{S,T}$ and $f_{S \bowtie T,R}$. From the probe orders σ_R and σ_T, we can observe $f_{R,S}$ and $f_{R \bowtie S,T}$ as well as $f_{T,S}$ and $f_{T \bowtie S,R}$. Now there are two additional cases for the predicates that influence how much we can learn from them. The first one is, if the predicate correlates with time. This means, as a probe from S to T will only join the arriving s tuple with tuples from T that arrived before, and if the predicate says "T comes after S", then the observed join $f_{S,T}$ will have close to zero results, however we cannot use this estimation for $f_{T,S}$. If the predicate is independent of time, then we can actually use the observed value of $f_{T,S}$ from σ_T to estimate the first part of σ'_S. The next case is independence of predicates. If $R.a = S.a$ and $S.b = T.b$ are independent, then $f_{R,S,T} = f_{R,S} \cdot f_{S,T}$. If this is the case, then we can estimate $f_{S \bowtie T,R}$ from $f_{T,S}$ and $f_{R,S}$ (both assuming time-independence).

Sampling

The data we gather for sampling depends on the query. For example, for a query $(R, S, R.a = S.a)$, the b-value (if it exists) of either relation is not interesting. Instead, for each relation R_i with attributes $a_1, \ldots, a_k$ occurring in some predicate of the query, we record a sample $R_i^S \subseteq \pi_{a_1,\ldots,a_k}(R_i)$ with sampling factor sf_i such that $|R_i^S| \cdot sf_i = |R_i|$.

As streams may occur with different input rates, it is not useful to set a sampling factor equal for each relation. For example, for a stream with low input rate it might be feasible to use the entire stream for the size estimation, however for a fast stream using the entire stream would overflow the sampling component.

7.2.2 Adapting to Changes

As data characteristics or query work load changes, a new strategy might become more optimal than the currently deployed one. The goal is to switch to this new strategy without downtime or loss of results in the meantime. We achieve this by dividing the time into epochs and making the configuration of all components depending on these epochs.

7.2.3 Epoch-Based Configuration

In Section 6.3, we introduced epochs as periods of time for grouping stored data. Now, we also use the level of epochs for dynamic reconfiguration. An epoch has a starting timestamp and is considered the current epoch until another epoch with a later time-

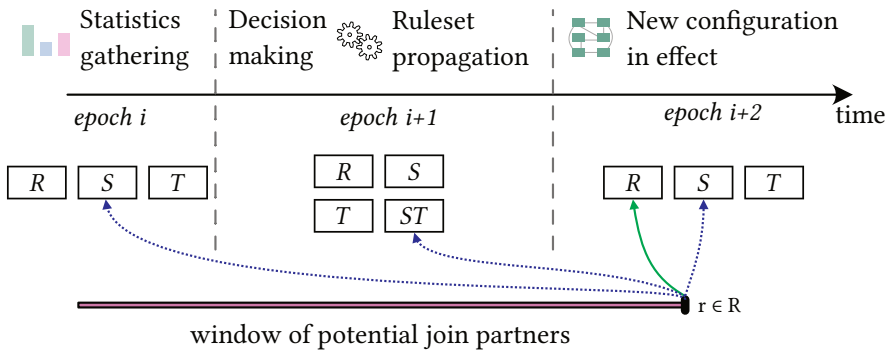


Figure 7.4: *Above:* Changes in statistics gathered during one epoch impact the epoch after the next one. *Below:* Tuple $r \in R$ arrives and can find join partners depending on the stores installed in the candidate epochs.

stamp is created. For each epoch, the data sampled from the epoch is used to create epoch-local data characteristics. This is done in the next epoch, and so changes can be decided for the epoch after that. Figure 7.4 illustrates this: during epoch i sample data is gathered from the inputs. When epoch $i + 1$ starts, the statistics from i are evaluated and fed into the ILP optimizer. If the optimization result differs from the previous one, a new configuration is created. This configuration is sent to all workers to be active starting at epoch $i + 2$.

In this example, there are two targets for the first probe, and in general there can be more. Thus, the task receiving the input tuple needs to keep track of where tuples need to be sent. Algorithm 7 demonstrates how this is done. In Line 2, we determine the target epochs where join partners according to the windows in the query can be. This also depends on the queries installed in the system. In Figure 7.4, for tuple r the target epochs are i, $i + 1$, and $i + 2$, and the receivers are the S and the ST-store. This could be the result of the optimizer deciding for epochs i and $i + 2$ to use probe orders $\langle R, S, T\rangle$ and for epoch $i + 1$ probe order $\langle R, ST\rangle$. In Line 3, we iterate over the receivers and then emit the tuple in Line 4 to the receivers and also send the target epoch. This epoch variable signals the state of the stores the probe tuple wants to see.

This is reflected in the changed handle function, also shown in Algorithm 7. Here each tuple arrives annotated with an epoch. Using this epoch, we get the ruleset that is valid for this epoch in Line 7. If there are store or probe rules, we also store or probe with respect to this epoch in Lines 10 and 11. This means, that also for each epoch, an independent container is created on each worker together with all aforementioned indexes. If at the end of probing a result is observed, the receivers of the next step depend on the epochs determined by the originating tuple's timestamp. In the end, the entire result consists of the union of the results of all covered epochs.

Algorithm 7 Adaptive version of handling procedures for tuples of input relations and intermediate tuples.

```
1  fun handle_input(tuple)
2    for epoch in get_epochs_for(tuple)
3      for receiver in receivers of target_epochs
4          emit(receiver, epoch, tuple)
5
6  fun handle(e_in, epoch, tuple)
7    rules ← ruleset[epoch, e_in]
8    for rule in rules
9      switch type(rule)
10         case StoreRule: store(epoch, tuple)
11         case ProbeRule: probe(tuple)
```

As the query's window is not aligned with the epochs, the workers need to check not only the join predicate, but also that the window condition is satisfied.

7.2.4 Supporting Query Changes

So far, the description focused on a given set of queries and how to adapt to changing data characteristics. In a long-standing streaming system, users also want to install new queries or remove old ones when they are not interesting anymore, which also captures updating a query. When a new query is installed, at the next run of the optimization procedure, it is also considered and corresponding probe orders will be generated. Hence, results can start being reported as soon as the new configuration is installed.

Typically, if a system starts answering a new query, the first window size does not contain all data. This is because only after the query is installed, tuples are started to be collected in operators for joining. Consider the scenario in Figure 7.5 where at time τ_0 a new query for joining R and S is installed. If streamed relations R and S were only to be observed since τ_0, consequently only these tuples can be probed against. This means, if at τ_1, a tuple from R arrives, and it is probed against the S-store, it is not possible for the system to match these tuples from S that satisfy the join predicate and the window condition, but were observable in the original data stream before τ_0, as indicated by the red line from τ_1 into the past. Vice versa for the tuple arriving at τ_2 which cannot meet the theoretical join partner from R. If at time τ_3 the tuple arrives and a join partner was arriving after τ_0 in the probed stream, this partial result can be made. Thus, only after waiting a full window length, such a system can provide complete answers. If a system is continuously running, and as it is answering other queries, the state used for the other queries is available to a new one. This means, the

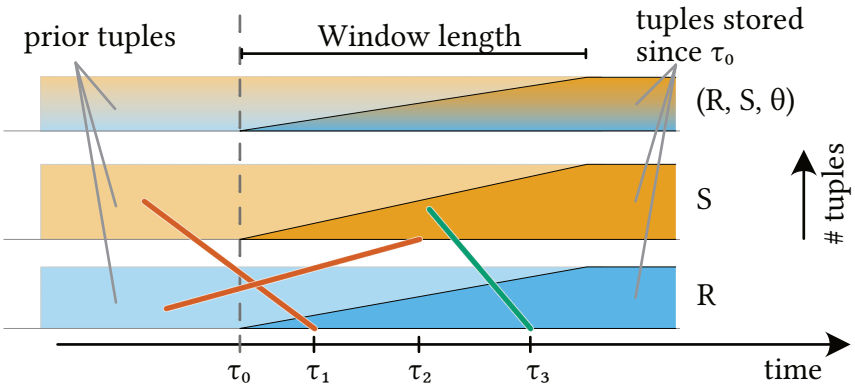


Figure 7.5: The result of query (R, S, θ) installed at τ_0 when it can only access tuples arrived later than τ_0 or also prior tuples.

registered stores can be exploited to provide complete answer for new queries more quickly.

If for all but one inputs of a query stores are registered, we compute probe orders for all epochs that overlap with the current window, and append these probe orders to the worker's configurations. This way, we can instantly begin answering all desired join results, and avoid the bootstrap problem of having incomplete statistics.

When a query is not needed anymore, it is removed from the optimizer input. But that also means, that previous store windows might not be needed anymore. A reference counting strategy determines the number of queries a store is serving. As soon as this counter drops to zero, the store is deregistered.

7.3 Experiments

The experiments are organized into three sections. First, we investigate the overall performance of our adaptive multi query optimization. Second, we specifically look at single-query performance to understand the benefits of adapting query plans to changing data characteristics. Last, the impact of input sizes to the ILP performance is investigated in detail.

We implemented the described routines as extensions to CLASH in Kotlin 1.4, and Gurobi 9.0.0rc2[1] is used as the solver for the ILPs. The optimized query plans are translated by CLASH into Apache Storm v2.2.0 [1] topologies and executed on OpenJDK 11 running on a compute cluster of 8 machines. Each machine has 128GB DDR3 memory and two Intel Xeon CPUs 1.7 GHz with 6 cores. This means, we could run up to 96 workers with 10GB memory in parallel. The cluster nodes are connected

[1] https://www.gurobi.com/

using 10Gbs ethernet network. Input data is consumed from and output is written to Kafka over the same network; the state of the stores is kept in the main memory of the worker processes.

7.3.1 Multi-Query Performance

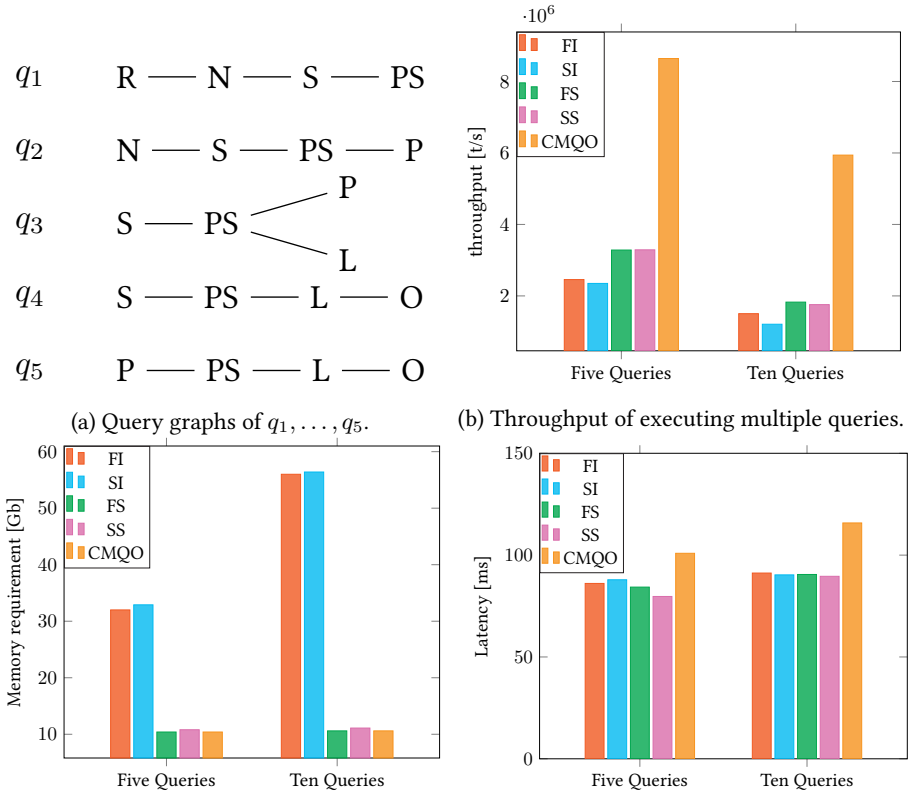
The following alternatives to processing bulks of queries are considered for comparison:

1. Several independent Apache Flink [8] Jobs, one for each query, are initiated. We refer to this strategy as **Flink Independent** (FI).
2. Analogously for Apache Storm topologies, coined **Storm Independent** (SI).
3. A naive multi query optimization strategy where each query is optimized individually with common subplans being executed only once and shared in Flink, coined **Flink Shared** (FS) and
4. likewise for Storm, **Storm Shared** (SS).
5. Lastly, our approach of global optimization: **CLASH-MQO** (CMQO).

Like in the previous chapters, we used the TPC-H data set [3] with a scale factor of 10. We create join queries based on present primary, foreign keys and, additionally, type compatible data of TPC-H, which means that two columns can be used for joining if they contain equal values. This leads to a mixture of common primary-foreign-key style joins, high-selectivity joins (e.g., on 'lineitem.linestatus' and 'orders.orderstatus' where the domain consists only of F, O, and P), and low-selectivity joins (e.g., on 'customer.custkey' and 'nation.nationkey' where only customer tuples with the lowest keys find a join partner). Using these potential joins, we construct queries by selecting a random relation and then randomly adding joins until the desired query size is reached.

We start by investigating the throughput of the systems. For this, data is fed into Kafka at the maximum sustainable rate for each configuration. The throughput is the time difference between the first and the last processed input tuple divided by the number of input tuples. We use the five queries shown in Figure 7.6a where no additional filtering was imposed on the inputs and the full history of the input tuples is considered, and another test with ten queries, with additionally more partly overlapping joins.

In Figure 7.6b we see the throughput of these workloads where Flink and Spark reach roughly the same performance as well as already a speed up of 1.4 with trivial sharing. Flink's throughput is a smidge higher what can be explained with the overhead of our routing implementation. Our approach of globally optimizing these



(a) Query graphs of $q_1, \ldots, q_5$.

(b) Throughput of executing multiple queries.

(c) Memory requirements for different query plans.

(d) Latencies of join computation of complete tuples.

Figure 7.6: Multi-Query Performance on TPC-H data.

queries brings us a speedup of 2.6 compared with the naive implementations. The great potential of sharing state can be seen in Figure 7.6c where we compare the Storm implementations of isolated and shared execution. We see here, that with five queries running independently, 3.1 times the memory is required and with ten queries even 5.3 times. For measuring the latency, we assign each tuple a timestamp when it arrives at the system and another timestamp when all join results with this tuple are computed, and record the differences between these timestamps. Figure 7.6d shows that the average latencies with shared multi query optimization are increased by 14 to 16%, compared to the other modes. This is due to the increased chance of selecting locally suboptimal probe orders which then leads to tuples taking longer in order to report a result.

7.3.2 Impact of Adaptation to Individual Queries

For this test we use a four-way linear join query of artificially generated relations $(R, S, T, U, R.a = S.a, S.b = T.b, T.c = U.c)$ where the inputs arrive with a constant rate of 100k tuples per second. The join attributes are set such that each tuple will be part of one join result, i.e., half of the tuples find join partners during probing. The window size is five seconds for each input and the epoch duration is one second. We initialize the optimizer with a little higher selectivity for $S(b), T(b)$ to make sure the probe orders $\langle S, R, T, U\rangle$ and $\langle T, U, R, S\rangle$ are selected.

We compare the latencies of adaptive reoptimization **(A)** and the initial static plans **(S)** and initially they perform very similar with a little short of 56ms latency, as depicted in Figure 7.7a. After 15 seconds the input changes drastically, now every tuple of S finds 100 join partners in R, but none in T; vice versa for T-tuples. Immediately after this the latency of both topologies increases slowly to about 72ms, which is due to tuples being longer in buffers as the workers try to catch up. In the adaptive strategy this works and after roughly a window a healthy latency is regained. The static strategy cannot recover from this change and eventually the workers failed due to memory overflow.

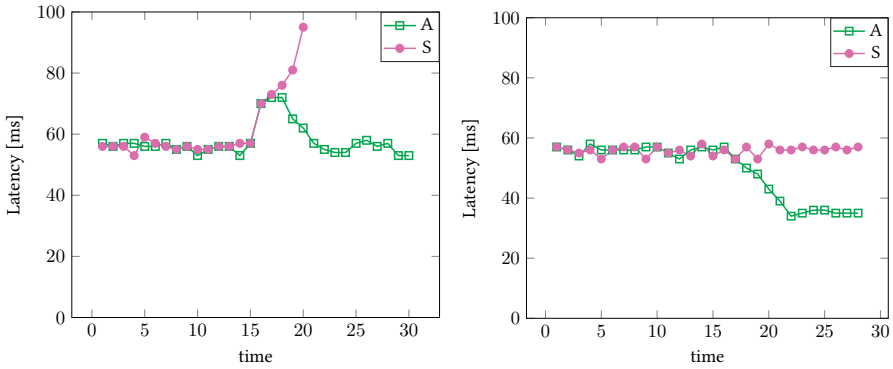
We then use the same query, but with different input rates. R has 5M tuples per second, the other ones several orders of magnitude slower at 5k tuples per second. In Figure 7.7b we show the latency for the static topology which remains at the same constant level. Again after 15 seconds we induce a change of the incoming data, now the size of the intermediate result of S, T, and U gets very low. This is recognized again after one epoch and a store for the result of the join of S, T, and U is introduced. We see a decline of the average latency that stabilizes from second 22 on a value of about 36ms. During the decline phase, join partners are found already in the new store, but also older join partners need to be probed iteratively.

7.3.3 ILP Optimization

We simulate an environment consisting of multiple relations that can be joined together with given input rates and join selectivities. In this environment we randomly generate queries and for each query we generate all probe orders and the corresponding ILP model as described in Section 7.1. This model is solved using Gurobi. We compare the cost of the joint query plan where query plans are shared and the cost for plainly applying only the best probe order for each query individually. Tests were conducted on a system with 3.1 GHz Intel Core i7 CPU and 16 GB main memory.

The input relations have all the same arrival rate and a join between any two relations has a selectivity of arrival rate^{-1}.

The first trial consists of ten input relations with three attributes each. We gener-



(a) A sudden increase in join selectivity renders a static join strategy unviable.

(b) Adaptive join processing lowers the average end-to-end latency.

Figure 7.7: Adaptive execution.

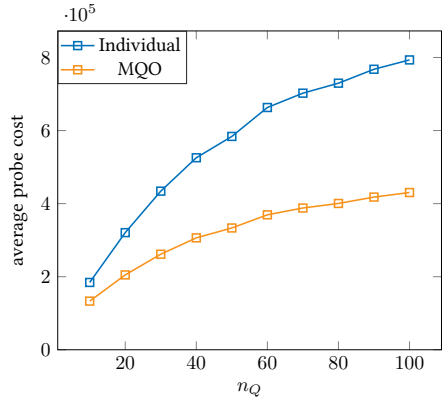
ate n_Q queries that each span three relations and eliminate exact duplicates (as these would be anyway answered together by a naive implementation). Figure 7.8a shows the cost without sharing in the line for **individual** optimization and with sharing for **multi query optimization**. The more queries we generate (over the same set of input relations), the higher the average probe cost gets for both. But in case of multi query optimization, the probe cost of the MQO is significantly lower, around 50%, than without sharing of probe order prefixes.

In Figure 7.8b, we show how the problem sizes grow: the number of variables fed into the ILP solver, indicated by the green line, grows more slowly the higher the number of queries; for 100 queries with each 3 relations, it is in average 1717. This slow growth is because the more queries are optimized simultaneously, the more potential for sharing probe order prefixes there is, and each shared probe order prefix also shares a variable. The purple line indicates the number of probe orders and it also grows slowly. This is due to the fact that as we draw more queries over the small amount of input queries, the chances of producing the same query again increases.

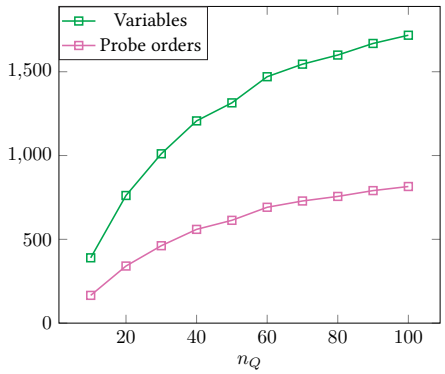
We now examine the benefits for a higher number of input relations: the queries are now randomly drawn from 100 input relations, each with three attributes. Figure 7.8c shows the probe cost savings, and here we see that for few queries nearly no savings are visible. For example, at 50 queries around 15% of the cost can be saved. In Figure 7.8d we see how also the problem size behaves more linearly. If we look at the absolute numbers, we see, for example for $n_Q = 50$ that 3000 variables are required compared to less than half of it in Figure 7.8b. This is due to the fact that the generated queries have very little overlap and thus only little possibility of sharing. Both graphs are not linear but slightly convex. This is because each new query also adds more possibilities for partitioning of a store, and each partitioning choice also increases the numbers of probe orders generated for a single query and consequently

also the number of variables generated in the ILP.

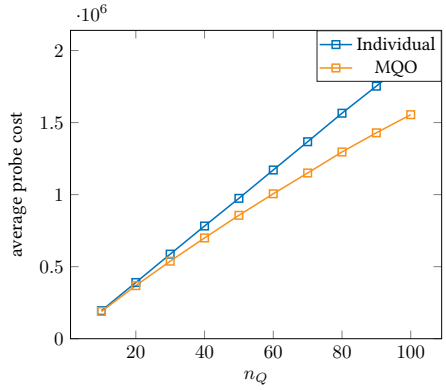
In Figure 7.8e, we show the runtime for optimizing a different number of queries generated over 100 input relations, and see that it grows linearly, while even at 100 simultaneous queries, the optimization time is at 120 milliseconds. In this experiment, all queries where over three relations. We wanted to find out, how this approach scales to bigger queries, and thus altered the query size, i.e., the number of relations input into a query. In Figure 7.8f, we see how the size of input relations effects optimization time. Already ten queries of size four take 400ms—one order of magnitude more than ten queries of size three. Optimizing ten queries of size five takes twelve seconds, and optimizing 30 queries of size five takes over two minutes. While this adds to the delay of restructuring the topology to run in an optimized way, query answering can begin earlier with locally optimized probe orders defined on the input relations.



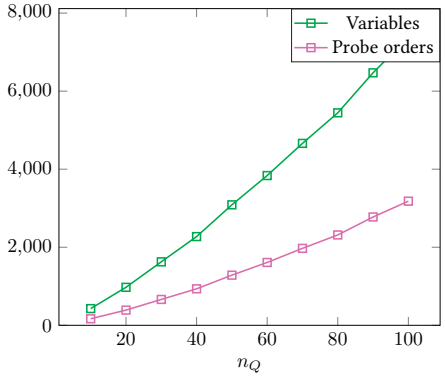
(a) Probe cost of queries over three relations, drawing from 10 input relations.



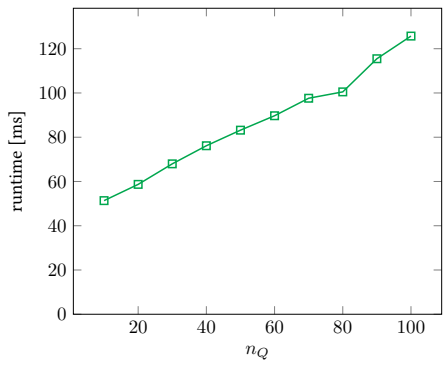
(b) Problem sizes of queries over three relations, drawing from 10 input relations.



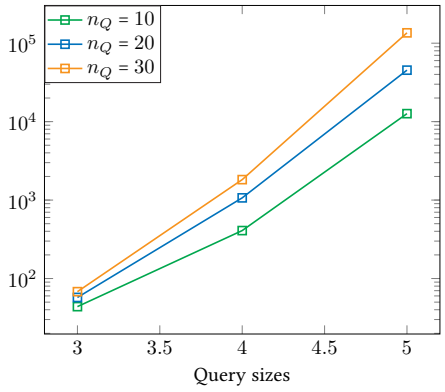
(c) Probe cost of queries over three relations, drawing from 100 input relations.



(d) Problem sizes of queries over three relations, drawing from 100 input relations.



(e) Runtime for different numbers of queries, drawing from 100 input relations.



(f) Runtime for different query sizes, drawing from 100 input relations.

Figure 7.8: ILP Experiments.

Chapter 8

The CLASH System for Multi-Way Join Computation

This chapter details on CLASH, the research prototype we built for implementing the join computation strategies presented in the previous chapters. CLASH is a generic and extensible system, and goes far beyond a proof-of-concept prototype. It accepts users' queries formulated in a subset of SQL or via a programmatic API. The optimization framework is designed such that experienced users can develop a custom optimizer, and the runtime can be extended by specially fitted local join algorithms.

CLASH is designed as high-level abstraction on top of Apache Storm, however, the optimization artifacts are abstract and can be applied to similar infrastructures that give control about tuple routing and stateful execution like the recent Apache Flink framework Stateful Functions [8].

CLASH is not reinventing the wheel when it comes to tuple routing primitives, like key-grouping, random assignment, full broadcast, etc., as it is using existing stream processors, as routing substrate—benefiting further from provenly robust, mature systems with out-of-the-box properties like fault tolerance. As it is focused on join computation, it only has very rudimentary support for other, orthogonal, operations like aggregation or classification.

We now first look at the big picture of CLASH and explain details of selected components in the following sections.

8.1 Overview

Figure 8.1 shows an excerpt of the different modules of CLASH and how they depend on each other; arrows point away from the depending module. The modules can be roughly divided into *core* and *runtime* components, where the former are responsible for understanding and optimizing queries and the latter are responsible for the execution of the query.

The **common** module contains the configuration and elementary types that are used by most other modules, like `Relation`, `AttributeAccess`, or `Predicate`. As it is automatically included in all other modules, these arrows are only lightly drawn for visibility. The **query** module contains concrete classes for describing relations and thus queries, and the SQL parser and the QueryBuilder which are used to construct query objects. The **physical_graph** module contains the `Physi-`

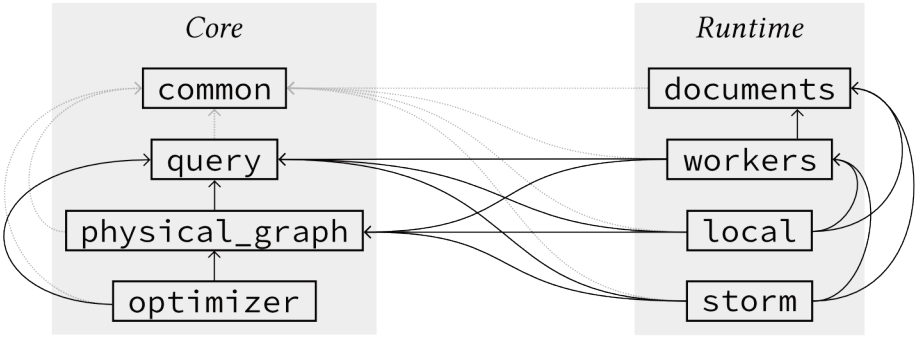


Figure 8.1: Excerpt of CLASH's modules and their interdependencies.

`calGraph` and a builder for it, and the **optimizer** module contains all optimizers. Since the result of the optimization phase is a physical graph, the optimizer module depends on physical_graph, and as both need an in-depth understanding of the query (relation/predicates) at hand, they depend on the query module.

In the runtime section, the **documents** module contains a description of the wire-format for messages that are sent between stores. The **workers** module contains the part of the stores that actually executes probing and storing operations along with indices and handling of late tuples. Note that this is separate from the **storm** module, which only cares about setting up bolts and spouts for a topology and using the rules to call correct functions of a worker. This loose coupling enables usage of other runtimes, like the one in the **local** module which emulates Storm in a single-threaded mode. As the storm and local modules need an understanding of `AttributeAccesses`, `Rules` and more, they depend on the query and physical graph modules; however they are not interacting with optimizers, and vice versa, optimizers do not care about the runtime logics.

There are more modules that are not shown in Figure 8.1 for clarity. These are tasked with connection to other systems (e.g. Kafka or Influx), with supporting frontend functions (like manager and strummer for a REST-API and web-interface), or contain test data like TPC-H queries and statistics.

8.2 Declarative Querying

Queries in CLASH are declarative; for example, although the syntactical order of input relations and predicates differs, the two SQL-like queries 8.2a and 8.2b are equivalent. In fact, these queries are also equivalent to queries 8.2c and 8.2d which are formulated using the programmatic `queryBuilder`.

While declarative querying is not new—in fact, every relevant database manage-

```
SELECT r.a, t.c
FROM r, s, t
WHERE r.a = s.a
  AND s.b = t.b
```

(a)

```
SELECT r.a, t.c
FROM t, s, r
WHERE s.b = t.b
  AND r.a = s.a
```

(b)

```
queryBuilder
  .from("R").from("S").from("T")
  .where("r.a = s.a")
  .where("s.b = t.b")
  .select("r.a", "t.c")
```

(c)

```
queryBuilder
    .from("S").from("T")
    .where("s.b = t.b")
    .from("R")
    .where("r.a = s.a")
    .select("r.a", "t.c")
```

(d)

Figure 8.2: Different forms of expressing the same query.

ment system supports this via SQL—the big contemporary stream processing systems do not or only poorly support this. The benefit of declarative querying is that neither human users nor query-generating programs accidentally encode a processing strategy into the query. Since users only express their intent, CLASH can freely choose a processing strategy best for the current data characteristics or available computing resources.

8.2.1 Representing Queries as Relations

In CLASH, streamed relations are represented by `Relation` objects. A `Relation` consists of

- **inputs**, a map from names of other relations to window definitions,
- **filters**, unary predicates that are defined on the input relations individually,
- **joinPredicates**, binary predicates that are defined on pairs of input relations,
- **projections**, a description of which attributes to include in the result, and
- an **alias**, a name for this relation.

This definition of a relation is purely a specification of the output tuples, and a query is a thin wrapper around the relation. So in the end, when a user submits a query, she defines how the resulting tuples should look like. CLASH's task is to generate a configuration of the underlying stream processor, to produce such tuples.

```
Relation(
    inputs = TODO(),
    filters = emptySet(),
    joinPredicates = setOf("r.a = s.a", "s.b = t.b").toBinary(),
    projection = Projection.AttributeProjection(
        listOf("r.a", "t.c").toAttributeAccess()
    ),
    alias = RelationAlias("q")
)
```

Figure 8.3: Construction of a relation object equivalent to the queries in Figure 8.2.

```
queryBuilder
    .from("part").from("partsupp").from("supplier")
    .where("part.partkey = partsupp.partkey".toBinary())
    .where("partsupp.suppkey = supp.suppkey".toBinary())
    .from("nation").from("region")
    .where("nation.regionkey = region.regionkey".toBinary())
    .where("supplier.nationkey = nation.nationkey".toBinary())
    .select("supplier.acctbal", "supplier.name" /*...*/)
    .build()
```

Figure 8.4: Using the QueryBuilder to create a query for TPC-H Q2.

Consider the constructor invocation shown in Figure 8.3. This creates a relation object equivalent to the previously discussed queries. Here it is again very explicit that no processing order is encoded into a query, as both, inputs and join predicates, are unordered. On the other hand, this set-oriented approach allows decomposing a relation into sub-relations, which in turn can be used by optimizers to better understand partial results they are creating.

Writing such relation objects directly is not very convenient, thus CLASH has two other methods discussed in the following two subsections, the QueryBuilder and an SQL-like interface.

8.2.2 Programmatic Query Builder

The recent trends on streaming APIs in programming languages, like Java and Rust, and data management frameworks, like Flink and Spark, show the urge to provide means to programmatically construct queries. For this, CLASH exposes the `QueryBuilder`, which can be used to define the components of a relation in a piece wise manner. Repeated method calls are used to register inputs, predicates, and projections, until the `build()`-method is called and constructs the desired `Relation` object.

```
queryBuilder
    .from("tweets", "newer", WindowDefinition.minutes(1))
    .from("tweets", "older", WindowDefinition.hours(1))
    .where("older.user_id = newer.user_id".toBinary())
    .select(Projection.StarProjection)
    .build()
```

Figure 8.5: Encoding window definitions while using the QueryBuilder.

A query over the well known TPC-H schema [3] representing the joins used in Q2 can be built as shown in Figure 8.4. Note, that the $n : m$ join between `part` and `supplier` is syntactically separate from the join between `nation` and `region` and thus both could be extracted into separate methods without sacrificing optimization potential. In the end, the optimizer is responsible for the push down of predicates as well as attribute selections.

Inputs and Windows

We differentiate between **input names** and **relation aliases**. An input name indicates the (technical) source of a tuple. This could be for example, a connection to Twitter collecting all tweets classified as Dutch-language tweets or a subscription to some Kafka topic. Such an input name needs to be registered independently of the query. A relation alias is a name that can be given to any relation, e.g., to query results. But we also use it to give inputs more meaningful names or for identifying multiple occurrences of the same input in a self-join scenario. For the writer of a query, assigning relation aliases is most of the time optional; CLASH implicitly uses the input name as relation alias if not specified otherwise.

Windows are assigned to relation aliases such that self joins can be conducted over different windows. Consider the query in Figure 8.5 with a self join of recent tweets (last minute) with older tweets (last hour) of the same user. Both `from`-calls refer to the same input name "tweets", and use different aliases and different windows.

8.2.3 The SQL Interface

In addition to the programmatic API, CLASH also offers a SQL-like interface. This interface is realized using the JSqlParser library which is aptly described on their project website as "JSqlParser parses an SQL statement and translate it into a hierarchy of Java classes. The generated hierarchy can be navigated using the Visitor Pattern".[1] While JSqlParser offers a bunch of features, like parsing DDL and DML statements

[1] https://github.com/JSQLParser/JSqlParser

```
SELECT *
FROM r('sliding', '1', 'minutes'),
     s('sliding', '2', 'minutes')
WHERE r.y = s.y
```

(a) Valid query

```
SELECT r.x, t.z
FROM r('sliding', '1', 'minutes'),
    s('5')
WHERE r.y = t.v
```

(b) Invalid query

Figure 8.6: Examples of a valid and an invalid query.

or being able to handle dialects of different DBMS vendors, we only use it to parse `SELECT`-queries or fragments of it.

Parsing (or rather, interpretation of parsing results) is part of the **query** module. Here, we define visitors for all features needed which collect information during their visit of the syntax tree nodes. Only the parts of the query that are relevant for CLASH are extracted, so for example, trying to parse `SELECT * FROM r WHERE r.y = (SELECT s.x FROM s)` will result in an exception informing the user that using a sub-select in an equals predicate is not supported.

Defining Windows

Standard SQL has no means of defining window restrictions on relations in the `FROM` clause, and our notion of a window is not to be confused with window functions that can be used in the `SELECT` clause for aggregation of values. Arasu et al. suggest in [11] a bracket notation like `FROM relation [RANGE 10 seconds] r` to indicate that the input relation `relation` should have a sliding window of size 10 seconds applied on and is available under alias `r` for the rest of the query. Supporting such a schema implies extending JSqlParser's lexer to understand such a bracket expression, which is possible but undesirable, as it means that updates of this library cannot be easily retrieved from upstream.

Instead, we formulate windows as table functions which JSqlParser supports out-of-the-box. The parsing result consists of the name of the table function and the arguments. We interpret the former as relation name and the latter as window definition. Consider for example the query in Figure 8.6a. Here, we have an input relation, `r` with a sliding window of 1 minute, and an input relation with a sliding window of 2 minutes. The first parameter of the table function indicates the type of the window (in this case, `sliding`), and the other parameters are parameters to the window.

Validation of Queries

While CLASH does not require a full-fledged schema, queries need to make sense. For example, the query like the one in Figure 8.6b has multiple problems. First, the projec-

tion list contains a reference to `t` which is not known; the same for the join predicate in the `WHERE` clause. Second, the arguments for table function `s` do not describe a window (that CLASH can understand). These are all reasons for the validation phase to reject this query.

8.2.4 Operations on Relations

A big benefit of such a declarative `Relation` object is, that it can easily be transformed into another one, similar to relational algebra operations. For example, two `Relation`s can be joined together by computing the union of their inputs, filters, join predicates, and projections, and by assigning a new alias.

Further, we can decompose a `Relation` into sub-relations as follows: Given a relation and a list of aliases, create a new relation with the inputs associated with the given aliases, all filters that reference one of the aliases, and only join predicates that compare two aliases. If the projection list of the parent relation contains explicit attributes, the sub-relation needs also to contain these as well as attributes referenced in join predicates that connect the sub-relation with the remainder. A new alias can be defined if necessary.

The relations given by a user are inputs and query results. Through composing and decomposing these relations, optimizers can understand which partial results are created and stored and access these partial results when answering a new query. Another way of thinking about this is as implicitly created materialized views in a database system.

Composability

A query can itself be input to one or multiple other queries. This way, a query written once can easily be reused. Using a subquery as input to another query is not even an optimization barrier, as such a complex query can easily be flattened: inputs and join predicates are unified and the projection is taken from the superquery.

8.3 The Physical Graph and the Optimizer

The goal of the optimization process is to generate a physical graph which is an abstract definition of a topology. While this work has Storm as a target in mind, as description of a data flow it is still abstract enough, that adapters to other systems can be used. We will first have a look at a physical graph, and then examine the optimizers which are used to create such a graph.

8.3.1 The Physical Graph

A `PhysicalGraph` is a graph consisting of **nodes** which are interconnected by **directed edges** and describes at a high level the processing strategy for answering a query. It plays a role similar to a physical query plan in a database system, hence the name.

Nodes can be of multiple types, for example, `SelectProjectNode`s represent combined selection and projection operations on streams, or `Store`s which store and probe tuples—some of the core components which we explained in the previous chapters. Like Storm bolts, these nodes carry a meaning of parallelism, and thus can be partitioned. For example, a `ThetaStore` is partitioned randomly, and a `PartitionedStore` is partitioned according to a list of attribute accesses.

Edges are labelled with a type, one of `ALL`, `SHUFFLE`, or `GROUP_BY`. If the receiving node of an edge is parallelized, this edge type indicates which parallel instances of that node are targeted. `ALL`-edges send a copy of the tuples to each instance, `SHUFFLE` a single copy to a randomly chosen instance, and `GROUP_BY` edges send tuples with the same group-by attributes to the same instance. This might seem redundant, since nodes can already be defined with a certain kind of partitioning (c.f. ThetaStore and PartitionedStore), however it is important to be able to send tuples with different types to the same store, for example for probing from two different relations as seen in the previous chapters.

In Section 4.2 we already discussed a simplified version of the physical graph for a join between two relations. Now we extend this example to see all details of the physical graph implementation as shown in Figure 8.7. On the left again are input stubs and on the right is the output stub. The input stubs are labelled with the relation they are serving, in this case only (R) and (S). In general, this could also be a relation with a filtering predicate, for instance, if the input comes from an API that allows such filtering, or even a more complex relation. In the center, there are two stores which are now labelled with the relation they store and also with the relation they produce. This reflects the twofold roles stores play, as storage for relations and execution for join operations. Not every store needs to produce a relation, for example for a three-way join with probe orders $\langle R, S, T\rangle$, $\langle S, R, T\rangle$, $\langle T, S, R\rangle$, only T- and R-store produce (R, S, T, θ). The nodes of the graph are connected with edges, which now not only have a type (in this case, shuffle or all), but also an automatically generated name $s_i, i \in \mathbb{N}$. This name is used to identify edges in rules. The R-input stub for example has two rules assigned which instruct it to send all incoming tuples of R over s_0 and s_3. The R-store has a corresponding role for receiving tuples over s_0 and store it as tuples from R. The S-store has the corresponding joinResult for tuples from s_3, which should be joined using predicate θ and use it as a result for the query. If a result is produced, the resultSend rule instructs this result to be sent along s_5. Note, that the

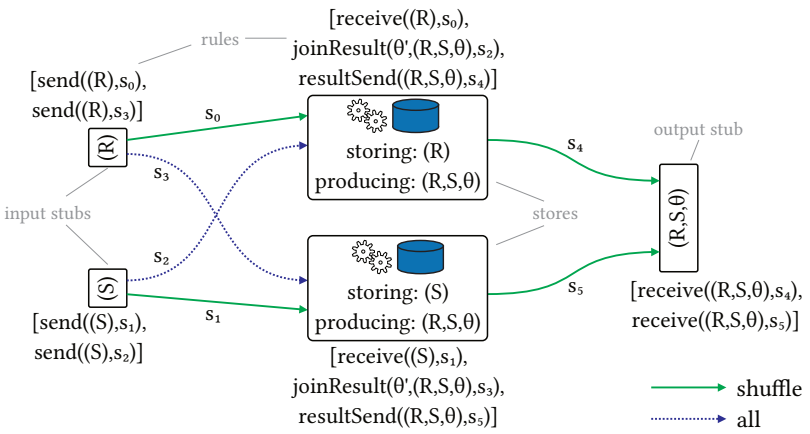


Figure 8.7: Physical graph with annotations for computing (R, S, θ).

joinResult rule is denoted with θ' instead of θ. This indicates, that this is not only the predicate itself but decorated with information on which input of the predicate is the store and which is the probe tuple. Finally, the output stub has receive rules attached to the incoming edges such that it understands which relations it receives.

8.3.2 Translating MultiStream Operators to Physical Graphs

In the chapters on MultiStream, we have seen how optimizers create different operator trees. Now these trees need to be used as templates to systematically construct a physical graph.

The physical graph is built from the bottom of the tree up. First, for each input, an input stub and a store are registered as well as connections between them. The input stubs are marked as "relation producers" for their corresponding input relation, which will be important later.

Then, for each MultiStream operator, the probe orders are implemented towards the top. The first step of the probe order $\langle \sigma_{R_1}, \sigma_{R_2}, \ldots \sigma_{R_n} \rangle$ is to connect the relation producers of σ_{R_1} with the store for σ_{R_2}. In case of a input relation these are the input stubs as mentioned before, but for materialized MultiStream operators, this might be a set of stores. The σ_{R_2} store gets an IntermediateJoinRule assigned that instructs it to react on incoming tuples of σ_{R_1} by probing with $\theta_{\sigma_{R_1},\sigma_{R_2}}$ and if there is a result, this should be sent to σ_{R_3}. These IntermediateJoinRules are used up to $\sigma_{R_{n-1}}$, for σ_{R_n} we assign a JoinResultRule. This rule does not consist of an output stream, only of the instruction for a final predicate evaluation. Instead, σ_{R_n} is also marked as relation producer for the result of this MultiStream operator.

Finally, all relation producers of the root node are connected with the output stub.

8.4 Join Processing Runtime

The previous sections were concerned with reading and optimizing queries, and generating a topology from that. This section explains, how the components that provide the runtime for join processing are built and deployed. We focus on the discussion of bolts which execute the store logic, but it should be noted for completeness that the runtime also includes spouts for reading several types of inputs, bolts for writing to several types of outputs, as well as more infrastructural elements for logging, watermarking etc.

8.4.1 Prefix Containers

A PrefixContainer is a thin layer that stores tuples in an inner data structure like a list or a hash map and probes that inner data structure. It is independent of Storm, such that it can easily be tested and also be deployed to other runtime providers.

The interface of a PrefixContainer basically consists of two functions, `store` and `probe`. The `store` function places a list of tuples for a given timestamp and attribute access into the prefix. The prefix may use this timestamp and attribute access for indexing, but this is a implementation decision. A list of tuples is used over a single tuple, as in general multiple tuples associated with the same timestamp are stored, as seen in the case of inner nodes of a join tree. This function also uses the given timestamp to find tuples that arrived as probes and have a higher timestamp in order to return a delayed join result as discussed in Section 5.4.1.

The `probe` function takes a list of tuples and predicate evaluations and returns a list of tuples, the join partners. Our predicate evaluations already carry the information which parameter of the predicate should be filled with the stored and which with the probed tuple. Again, this probe also takes timestamp and attribute access as parameters, so the implementation can use this for index access. The implementors of this function are responsible for saving the probe tuples for the delayed result generation.

8.4.2 Store Bolts

The StoreBolt is now the shell around a (or multiple) prefix container(s). As the name suggests, it is a Bolt in the sense of Storm, so it implements the `execute` function that reacts on receiving Storm tuples. In CLASH, the StoreBolt contains the set of rules that were assigned to the corresponding physical graph nodes.

Algorithm 8 shows in a condensed way the implementation of the execute function. It receives a (Storm) tuple and the sourceStreamId over which this tuple arrived. This sourceStreamId is used to lookup the rule which was installed in the physical graph for handling tuples from the corresponding edge. For the rule, there are three

Algorithm 8 Pseudocode for the execute method of a StoreBolt.

```
input: stormTuple, sourceStreamId
1  rule = rules.get(sourceStreamId)
2  when (rule) {
3      is RelationReceive -> store(stormTuple)
4      is IntermediateJoin -> probe(stormTuple, rule.target)
5      is JoinResult -> probe(stormTuple, rules.sendRulesFor(rule.relation))
6  }
```

important distinctions: it can be a relation receive, intermediate join, or join result rule. In Line 3, the receive rule means, that the arriving tuple belongs to the relation that is located on this store, so the contained documents are extracted and stored into the prefix container. If it is an intermediate join rule (Line 4), the tuple is probed against the prefix and the results (if any) are sent to the targets set in the current rule. In the case of a join result rule (Line 5), the tuple is again probed against the prefix, but the targets need to be found in the rule set.

This rule-based logic is necessary to allow flexible customization even at runtime. Further, new functionality can be added easily by adding new types of rules. For example, if a programmer wants to implement optional persistence, a `Flush` rule that forces a write of the stored intermediate results can be added to the store bolt without interfering with the already existing rules.

Chapter 9

Conclusion and Outlook

In this work, we have presented CLASH, a full-fledged approach to processing theta joins in a multi-way fashion. For this, we developed a novel n-way join operator, coined MultiStream, that leverages flexible probe orders of incoming tuples and allows constructing versatile join trees. Inside these trees, through an adjustable amount of specific materialization points, trading off between network bandwidth consumption and storage requirements is enabled and can be harnessed through the described cost models and cost-based optimization. The presented approach is extensible in a way that it allows other operators, like the join-matrix–based HyperCube approach, to be be easily integrated. We have shown that the integration of more efficient routing strategies for equality predicates with the less efficient but more powerful theta-join computation is feasible and that a single system can answer such queries.

Further, for computing multiple queries at once, we have seen significant improvement in throughput when combining multiple queries into one big topology. The increase in throughput means, that more tuples can be executed in the same time, thus the same compute cluster can process streams with a higher arrival rate. Static join ordering, like used in all currently available streaming systems, is prone to changes in the size of intermediate join results. However, a strategy for adopting to such changes avoids crashes, expensive recovery, or missing results.

Both our optimization approaches are computationally expensive, so they are meant to be used in a scenario like stream processing, where higher upfront cost of installing a query can be tolerated since the query is meant to be long-running. For our ILP-approach, optimization takes least time if the individual queries are smaller. Up to 30 queries of size five can be optimized within a second, which is still very usable for streaming scenarios.

With CLASH, we showed that our approaches are functioning in a realistic system where complex operators are set up for query answering in an automated fashion. Ultimately, join operations are part of the infrastructure of a data analytics application, and perform in tandem with other operators like aggregations or data access. Our usage of stores as containers of entire relations and designated sets of relation-producers enable the extension of CLASH or the adaption of MultiStream into other systems.

Outlook

We envision that this general approach to compute arbitrary join queries over multiple streamed relations has the potential to enable new, scalable streaming analytics tasks

which were not done so far because no system supported it.

In the future, systems could incorporate MultiStream's principles to enable theta-join computation, and thus extend the types of queries they support. Also, the usage of multi-way trees can, even without theta-join support, be helpful in answering queries on memory-limited systems, due to the memory optimality of flat trees.

In this thesis, we looked at tuple-by-tuple computation, with the relaxation on multiple tuples at once if they are intermediate results of a single input tuple. For high-rate streams there might be use-cases where batching up sequences of input tuples might be beneficial, e.g., for sharing iterations in the nested-loops join during probing and limiting the number of total communication events. This approach is for example pursued by Spark's Streaming API [10].

A limitation of our approach is, that there is no support for truly n-ary predicates that are not conjunctions of binary predicates. An example for such a predicate is $\theta(r, s, t) = (r.a + s.b = t.c)$. Here, tuples of all three input relations $r \in R$, $s \in S$, and $t \in T$ are required to evaluate if (r, s, t) belongs to the join result. With the current setup of MultiStream, the answering of such a query requires computation of a Cartesian product which we try to avoid. However, there might be other ways to cope with such n-ary predicates.

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] Apache Storm. http://storm.apache.org/. Accessed on: 15 November 2016.

[2] Joins | Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/table/sql/queries/joins/. As of commit ae813a7397955d7694109c3455b5cca14ac37938 of https://github.com/apache/flink from 24 November 2021.

[3] The TPC-H benchmark. http://www.tpc.org/tpch/. Accessed on: 18 November 2016.

[4] F. N. Afrati, M. R. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround distributed join algorithm. In M. Benedikt and G. Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, volume 68 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[5] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

[6] M. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[7] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 577–588. ACM, 2013.

[8] Apache Software Foundation. Apache Flink. https://flink.apache.org/. Accessed on: 8 August 2018.

[9] Apache Software Foundation. Apache Kafka. https://kafka.apache.org/. Accessed on: 8 August 2018.

[10] Apache Software Foundation. Apache Spark. https://spark.apache.org/. Accessed on: 8 August 2018.

[11] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[12] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In G. Weikum, A. C. König, and S. Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 419–430. ACM, 2004.

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In L. Popa, S. Abiteboul, and P. G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16. ACM, 2002.

[14] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[15] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In R. Hull and W. Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284. ACM, 2013.

[16] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 975–986. ACM, 2010.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[18] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 203–214. Morgan Kaufmann, 2002.

[19] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 40–51. ACM, 2003.

[20] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In E. A. Brewer and P. Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[21] R. Derakhshan, A. Sattar, and B. Stantic. A new operator for efficient stream-relation join processing in data streaming engines. In Q. He, A. Iyengar, W. Nejdl, J. Pei, and R. Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 793–798. ACM, 2013.

[22] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In A. Pirotte and Y. Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 151–164. Morgan Kaufmann, 1985.

[23] T. Dökeroglu, M. A. Bayir, and A. Cosar. Integer linear programming solution for the multiple query optimization problem. In T. Czachórski, E. Gelenbe, and R. Lent, editors, *Information Sciences and Systems 2014 - Proceedings of the 29th International Symposium on Computer and Information Sciences, ISCIS 2014, Krakow, Poland, October 27-28, 2014*, pages 51–60. Springer, 2014.

[24] M. Dossinger and S. Michel. Scaling out multi-way stream joins using optimized, iterative probing. In C. Baru, J. Huan, L. Khan, X. Hu, R. Ak, Y. Tian, R. S. Barga, C. Zaniolo, K. Lee, and Y. F. Ye, editors, *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*, pages 449–456. IEEE, 2019.

[25] M. Dossinger and S. Michel. Optimizing multiple multi-way stream joins. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1985–1990. IEEE, 2021.

[26] M. Dossinger, S. Michel, and C. Roudsarabi. CLASH: A high-level abstraction for optimized, multi-way stream joins over apache storm. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1897–1900. ACM, 2019.

[27] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 3rd Edition.* Addison-Wesley-Longman, 2000.

[28] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[29] J. Fang, R. Zhang, X. Wang, and A. Zhou. Distributed stream join under workload variance. *World Wide Web*, 20(5):1089–1110, 2017.

[30] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In J. C. Freytag, P. C. Lockemann, S. Abiteboul,

M. J. Carey, P. G. Selinger, and A. Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 500–511. Morgan Kaufmann, 2003.

[31] J. S. Gomes and H. Choi. Adaptive optimization of join trees for multi-join queries over sensor streams. *Information Fusion*, 9(3):412–424, 2008.

[32] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 146–155. IEEE Computer Society, 2007.

[33] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Query processing of multi-way stream window joins. *VLDB J.*, 17(3):469–488, 2008.

[34] M. Henderson and R. Lawrence. Are multi-way joins actually useful? In S. Hammoudi, L. A. Maciaszek, J. Cordeiro, and J. L. G. Dietz, editors, *ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems, Volume 1, Angers, France, 4-7 July, 2013*, pages 13–22. SciTePress, 2013.

[35] M. Hoffmann and S. Michel. Scaling out continuous multi-way theta-joins. In F. N. Afrati and J. Sroka, editors, *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2017, Chicago, IL, USA, May 19, 2017*, pages 8:1–8:4. ACM, 2017.

[36] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 761–772. ACM, 2007.

[37] J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. B. Zdonik. High-availability algorithms for distributed stream processing. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 779–790. IEEE Computer Society, 2005.

[38] A. E. Ingham. A tauberian theorem for partitions. *Annals of Mathematics*, 42(5):1075–1090, 1941.

[39] H. Jafarpour and R. Desai. KSQL: streaming SQL engine for apache kafka. In M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi,

editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 524–533. OpenProceedings.org, 2019.

[40] M. Joglekar and C. Ré. It's all a matter of degree: Using degree information to optimize multiway joins. In W. Martens and T. Zeume, editors, *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, volume 48 of *LIPIcs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[41] A. Jonathan, A. Chandra, and J. B. Weissman. Multi-query optimization in wide-area streaming analytics. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 412–425. ACM, 2018.

[42] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 607–622. ACM, 2019.

[43] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. *PVLDB*, 11(11):1332–1345, 2018.

[44] W. Lang, R. V. Nehme, and I. Rae. Database optimization in the cloud: Where costs, partial results, and consumer choice meet. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[45] Q. Li, M. Shao, V. Markl, K. S. Beyer, L. S. Colby, and G. M. Lohman. Adaptively reordering joins during query execution. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 26–35. IEEE Computer Society, 2007.

[46] R. Li, M. Riedewald, and X. Deng. Submodularity of distributed join computation. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1237–1252. ACM, 2018.

[47] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 811–825. ACM, 2015.

[48] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 227–230. IEEE Computer Society, 2017.

[49] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. Data ingestion for the connected world. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[50] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-store: Streaming meets transaction processing. *PVLDB*, 8(13):2134–2145, 2015.

[51] D. Oguz, S. Yin, A. Hameurlain, B. Ergenc, and O. Dikenelli. Adaptive join operator for federated queries over linked data endpoints. In J. Pokorný, M. Ivanovic, B. Thalheim, and P. Saloun, editors, *Advances in Databases and Information Systems - 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*, volume 9809 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2016.

[52] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 949–960. ACM, 2011.

[53] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205. IEEE Computer Society, 2016.

[54] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[55] R. P. Stanley and S. Fomin. *Enumerative Combinatorics*, volume 2 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1999.

[56] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014.

[57] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[58] G. van Dongen, B. Steurtewagen, and D. V. den Poel. Latency measurement of fine-grained operations in benchmarking distributed stream processing frameworks. In *2018 IEEE International Congress on Big Data, BigData Congress 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 247–250. IEEE Computer Society, 2018.

[59] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 37–48. ACM, 2002.

[60] A. Vitorovic, M. Elseidy, K. Guliyev, K. V. Minh, D. Espino, M. Dashti, Y. Klonatos, and C. Koch. Squall: Scalable real-time analytics. *PVLDB*, 9(13):1553–1556, 2016.

[61] A. Vitorovic, M. Elseidy, and C. Koch. Load balancing and skew resilience for parallel joins. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 313–324. IEEE Computer Society, 2016.

[62] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. Building a replicated logging system with apache kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, 2015.

[63] S. Wang and E. A. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In M. L. Kersten, B. Novikov, J. Teubner, V. Polutin, and S. Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 299–310. ACM, 2009.

[64] X. Wang, C. Jiang, J. Fang, X. Wang, and R. Zhang. Adaptmx: Flexible join-matrix streaming system for distributed theta-joins. In J. Pei, Y. Manolopoulos, S. W. Sadiq, and J. Li, editors, *Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21-24, 2018, Proceedings, Part II*, volume 10828 of *Lecture Notes in Computer Science*, pages 802–806. Springer, 2018.

[65] J. Yang, Y. Zhang, J. Wang, and C. Xing. Distributed query engine for multiple-query optimization over data stream. In G. Li, J. Yang, J. Gama, J. Natwichai, and Y. Tong, editors, *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part III, and DASFAA 2019 International Workshops: BDMS, BDQM, and GDMA, Chiang Mai, Thailand, April 22-25, 2019, Proceedings*, volume 11448 of *Lecture Notes in Computer Science*, pages 523–527. Springer, 2019.

[66] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.

[67] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning. In M. Lee, K. Tan, and V. Wuwongse, editors, *Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12-15, 2006, Proceedings*, volume 3882 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2006.

Lebenslauf

Manuel Dossinger[1]

seit 2015	**Promotion: TU Kaiserslautern** Fachbereich Informatik, Prof. Michel *Thema:* Optimizing Multi-Way Joins for Adaptive, Scale-out Stream Processing.
2013 – 2015	**Master of Science, Informatik: TU Kaiserslautern** *Abschlussarbeit:* Algorithms and Probabilistic Models for Similarity Joins over JSON Documents *Vertiefung:* Informations- und Kommunikationssysteme
2009 – 2013	**Bachelor of Science, Informatik: TU Kaiserslautern** *Abschlussarbeit:* Incremental Iterative Graph Algorithms with MapReduce *Vertiefung:* Informationssysteme
1999 – 2008	**Abitur: Theodor-Heuss-Gymnasium, Ludwigshafen am Rhein**

[1] geb. Manuel Hoffmann