Technical University of
Kaiserslautern

Department of Computer Science

Master's Thesis

Late Materialization for Multiway Stream Joins

Author:
M Tahmid Ekram

Submitted on:
18.02.2020

Supervisor:
MSc. Manuel Dossinger

Reviewers:
Prof. Dr.-Ing. Sebastian Michel
MSc. Manuel Dossinger

Abstract

Recent improvements in real-time data processing technology has provided a strong platform for quick and accurate analysis of high volume streaming data. Combining multiple streams via join operations in order to obtain more complex insights is very common in both business and scientific applications. We introduce a late materialization model for computing join results in stream join processing that is capable of improving performance by avoiding unnecessary communication. The approach includes generalized cost models which allow cost prediction before hand. This provides great versatility to the application in terms of being able to dynamically choose between early and late materialization based on input data characteristics. We provide theoretical cost analysis for various input cases and experimental evaluations for assessing the performance of the late materialized approach in stream join computation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in technology, primarily in the areas of affordability, portability and accessibility, are now allowing more people to be part of the digital age then ever before. The number of active cell phones in the world is almost equal to the total population [5] on earth and around half of these active cell phones are smart phones [6]. Even devices that were once considered as niche such as digital assistants (Amazon Alexa or Google Home) are now becoming commonplace. Though it may seem that these things are all independent and serve their own purpose, they are actually connected by the one common thing they generate; data.

An unimaginable amount of information is generated everyday in the world by people using technology. The rate at which data is generated is so fast that it becomes impossible to store and process it in a traditional manner. Let's take an example of what happened in one internet minute across popular online services in 2019 [9]:

- 4.5 million videos were viewed on YouTube,
- 3.8 million searches were performed on Google,
- Over $900,000 were spent on online shopping and much more.

All these activities generated vastly useful information. Every view on a YouTube video needs to be considered for services such as compensating and promoting the creator and computing general statistics. Online shopping websites such as Amazon will use click tracking to track the user's movement around a purchase to monitor its legitimacy and also advertise and promote products according to the user's viewing patterns.

These kinds of data that are generated constantly and in large volumes are classified as *data streams*. There are many sources which produce information in the form of data streams [10]. Sensors from industrial equipment and vehicles send data in the form of streams to applications, which monitor the performance and attempt to detect any potential defects in advance. Stock market changes also generate pricing information real time and financial institutions can use this to predict changes and adjust their portfolios. Location data streamed from consumer mobile devices allow manufacturers to provide smart insights in advance such as traffic or weather conditions. Online machine learning is another very popular application of streaming data [21]. Machine learning models are initialized and then constantly updated based on feature changes within the data stream.

Many options exist when considering how to work with streaming data. Building custom streaming applications is possible with the help of proven frameworks such as Apache Storm [4], Apache Kafka [2] and Apache Flume [1]. Cloud platforms such as Amazon Web Services also provide more advanced products

such as Amazon Kinesis [3] which have several additional features such as real time dashboards and alert generators built in. These allow users to intuitively build specialized streaming applications.

1.1 Result Materialization

The term result materialization refers to the time point or stage within a query plan where the actual result or results of the query are fully observed. Within the query plan there maybe several partial results which are used for further steps. An example would be the hash table made up of hash values from the join key columns used primarily for hash join algorithms. Another example, specific to column stores, would be position lists of tuples from a given column that satisfy the provided join predicate. Only when the complete result tuple is observed we can say that it is fully materialized. Using full tuples as soon as they become available is called the early materialized approach. But when the result materialization is delayed to the very end it is usually referred to as late materialization. If materialization takes place somewhere in between it is partial late materialization [15].

1.2 Problem Statement

Application of late materialization has potential to improve performance in stream join processing applications. In this regard, we present a late materialization model for join processing in data streams that aims to improve performance by reducing communication and storage costs during join result computation. The model works by calculating potential communication costs for both eager and lazy materialization in order to decide between the two. Our approach for late materialization of results in stream join processing is based on common concepts seen in contemporary stream processing frameworks [16, 18, 19] in order to ensure adaptability.

1.3 Contributions

- In this thesis we introduce a *late materialization model* applicable to general stream join processing frameworks. In order to optimize storage and communication, the model dictates that any intermediate join results that are being stored will only contain the fields that are needed to compute further joins.
- Intermediate result stores will work with join attributes only, be it for the purpose of storing or forwarding. Given a distributed setup for the streaming application, this allows for reduced network traffic and overall faster processing within worker nodes.
- Once an entire join plan has been fully traversed, only the join attributes and primary identifiers of the streams are available at hand. In order

to construct the complete result, our model will contain a *reconstruction phase* which involves fetching the relevant tuples from each relation's store and adding it to the result. This is similar to the column re-access performed in column stores.

- We will also propose methods by which the order of this reconstruction can be optimized in order to minimize network traffic. Since late materialization may not always be beneficial, we will also present certain criteria which can be used by the stream processing framework to decide between eager and lazy materialization.
- Our model dictates how the intermediate results are treated and also the final reconstruction order. It is assumed that the stream processing framework will provide an optimum join plan for queries involved.

1.4 Outline

Further chapters in this document are arranged as follows. Chapter 2 provides thorough background knowledge on all concepts and tools that have been used to create and test the late materialization models. CLASH [16], which we will use as a base framework for conceptualizing and testing our late materialization model, will be discussed in detail. Further clarification of result materialization and its adaptation to streaming joins will also be discussed. In Chapter 3 we cover some recent work that has been done the field of stream join processing. This will include few streaming join algorithms and frameworks including like CLASH. In addition we will also discuss some applications of late materialization in join processing with regards to column stores and how it can be adapted to row oriented stores. In Chapter 4 we elaborate our approach in detail by using a sample *Base Topology* as an example. We provide details on costs incurred with our base topology in terms of bytes stored/communicated for both lazy and eager materialization followed by a more general cost model. The cost models allow for two important things, showing the effectiveness of late materialization and to provide a metric for choosing between early and late materialization. Chapter 5 contains implementation details for the experimental model we have used. Evaluation of the model, both theoretical and experimental are detailed in Chapter 6.

Chapter 2

Background

Data streams can be processed in various ways to obtain useful insights from them [17]. Anomalies or unusual occurrences can be detected on continuous streams to find possible sources of error or defect. This is comparatively a simple process because it only involves finding out if a particular item deviates significantly from the expected distribution. Aggregation, grouping and clustering are also common operations performed on data streams. Updating the average temperature every time new readings arrive from sensors is an example of continuous aggregation. Performance statistics streamed in from factory machinery can be clustered based on factors such as operating temperature, operating speed, error count etc. These clusters can, after a certain period of time, be analyzed to find groups of machines that are behaving incorrectly.

Combining multiple streams can sometimes yield interesting results. When working with relational databases join operations are the most common approach for combining multiple tables. This is especially true for star schema based stores where very complex analytic queries involve mostly conditional join statements between fact and dimension tables. Join operations are important and also common place when querying relational databases. The same is also true when running queries on data streams. It is crucial for combining information from different streams, related or even unrelated [13]. Network monitoring is a use case for joining streams. Streams of packets flowing through two different routers can be joined on packet ids, making it possible to identify the packets which flowed through both routers and then measure certain statistics such as how much time on average packets take to travel from one router to another. Online real time auctioning systems can also make use of streaming joins. We need to join the stream of opening auctions with the stream of bids to identify which bids are for which auction. Since this operation can run at a massive scale (world wide bidding) it is important to ensure scalability and fault tolerance. So, it is of no surprise that a lot of work is done investigating novel and more efficient approaches to joining data streams [16, 18, 19]

Due to the unbounded and high velocity nature of streaming data, it is very important to design streaming applications such that these can adapt well. Two major problems, especially for join computation in streams, that appear most often are storage and processing speed.

- *Storage* - It might not be obvious at first as to why storage is an issue in join processing for data streams. For example in cases were operations are performed on a fixed window of tuples storage is not that much of a problem. This is because only the tuples within a window need to be stored and processed. When we consider full history joins, like we do in this thesis, the case is much more different as it is required for all observed tuples to be persistently stored. It is also very common for different data streams to have different arrival rates. One stream may generate tuples

very fast while the other one might be quite slow. To guarantee that most if not all join results are observed, each tuple should ideally be available as long as it has not successfully found its join partner. This is almost never possible because storage is always finite. So minimizing the amount of information stored for processing streaming joins is a possible approach to having increased capacity and efficiency in streaming applications.

- *Processing speed* - Keeping up with the arrival rate of streaming relations is another big challenge for stream processing applications. If the system is not scalable enough, it will not be able to adapt to high tuple arrival rates. The system will be at peak capacity by the items already being processed. Once the input buffer is exhausted new incoming tuples will simply be dropped. This is extremely problematic for join computations because tuples that have been dropped could have been part of potential join results. On the other hand it also does not make much sense to maintain a plethora of server resources because tuple arrival rate may slow down in certain situations (e.g. bidding stream on an unpopular auction) which will lead to a waste of resources.

In order to tackle said issues, a common method is to design and implement stream join processing algorithms which are scalable and can operate in highly distributed environments [16,18,19]. In accordance to this most streaming application development frameworks also come with support for such non functional requirements. Approximate query processing for data streams [14,23] is another potential method. Calculative load shedding can be used to drop tuples explicitly while still making sure that the results obtained are at an acceptable level of accuracy. The approximated result should be a very close representation of the actual full result.

2.1 Apache Storm

Many stream join processing frameworks [16, 19] are built on Apache Storm [4]. Storm is a real time distributed stream processing framework. It offers many of the features that are essential when working with real time distributed streaming operations such as scalability, fault tolerance and guaranteed message processing. The framework's core design and working approach is simple to understand and use making it very popular among researchers as well as industry pioneers [7].

Storm's core abstraction is based upon the concepts of streams, spouts and bolts. These all combine to form Storm topologies which can then be deployed on a Storm cluster.

- **Streams**: Stream is a very important abstraction in Storm. Streams are created using a unique stream ID. Streams transport tuples by using defined *Fields*. Each field is representative of an attribute of the tuple. To emit a tuple it must first be constructed using the field values and then optionally assigned to a stream. If no stream assignment is mentioned, the tuple is emitted over Storm's *default* stream.

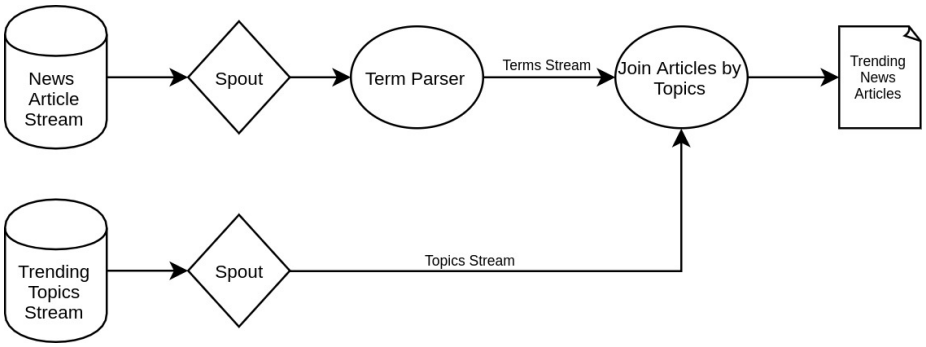


Figure 2.1: Illustration of a Storm topology

- **Spouts**: Spouts are points where streaming data enters the topology. Usually spouts will read tuples from an external source such as the Twitter API or data from sensor networks. These are then mapped to streams and then forwarded on to bolts for further processing. Spouts are capable of emitting multiple streams with different stream IDs.
- **Bolts**: Bolts are the main processing units in Storm. They receive incoming tuples from spouts and perform various operations on them such as filtering, joining, aggregation, writing to databases etc. Bolts can forward partially processed information to other bolts for further processing.
- **Topologies**: A Storm topology (Figure 2.1) is a network of interconnected spouts and bolts. This is equivalent to a MapReduce job in Hadoop. But the main difference is that MapReduce jobs end after processing is over but topologies will run indefinitely unless these are explicitly killed.

Storm topologies are deployed on Storm clusters. Clusters usually have a master node and several worker nodes. The master node monitors the worker nodes and distributes the topology code among workers. Each spout or bolt can have multiple running instances called **tasks**. And these tasks can be distributed among several worker nodes. How the incoming tuples are distributed among the tasks can be defined using specific kind of **groupings**. Figure 2.1 illustrates a Storm topology. Here the topology ingests data from two input streams. One input streams in the latest news articles and the other input streams the latest trending topics. The *Term Parser* bolt tokenizes the news article and outputs a stream of terms along with the news article identifier. More functionality such as removal of stem words can also be added to this bolt. On the other hand we can have an entirely separate bolt for handling stem terms. How much load is put on a bolt depends on how it performs. If a bolt is being overworked when the topology is running it is a good idea to separate its tasks into separate bolts to distribute the work load. Several tasks of a bolt can also run in parallel to allow for scalability. The *Join Articles by Topics* bolt takes the terms from news articles and trending topics as input and then joins these based on the term. A result of such a join denotes news articles talking about trending topics. In order to test the late materialization model, we have implemented a topology using Storm to perform join operations using both late and early materialization in order to compare the two.

2.2 CLASH

CLASH is distributed stream processing framework based on the MultiStream operator presented by Dossinger et al. [16]. It is a high level abstraction built on top of Apache Storm. It is capable of taking SQL queries as input and then optimizing and translating them into Storm topologies. The MultiStream join (MSJ) operator is capable of taking multiple relations and computing join results. It is also possible to materialize some intermediate join result and feed this into another MSJ operator, essentially forming a tree of MSJ operators. This allows the possibility of several join plans based on the number of relations involved and also which intermediate results are materialized. Investigating several such plans and choosing the optimum one is also a part of CLASH. Users provide connection parameters to data sources along with join queries over those sources as input to the system. Statistics such as selectivity and arrival rate of tuples can also be provided. These are useful for creating an optimal query plan and also deciding on certain options such as whether to use late materialization or not. The queries are then optimized and translated into Storm topologies on fly. Once the topology is ready it is deployed and started within the CLASH infrastructure itself.

An important abstraction in CLASH is the concept of *stores*. Stores are essentially Storm bolts which are capable of computing joins and storing intermediate results. Number of stores within the topology depends on the join plan. Each relation involved in the user submitted queries gets its own store and each intermediate join result that is required by the join plan also gets its own store. Upon arrival, the tuple of any relation will be first saved in its corresponding store and it will then probe the other stores in order to find a join partner. The order in which the remaining stores are probed is optimized based on the selectivity values provided by the user.

For linear iterative probing, all of the relation stores need to be probed to find the join result when a new tuple arrives. This can be avoided by having intermediate results materialized (Figure 2.2). It is possible to use multiple MSJ operators in order to construct join trees. This can be done by materializing one or more intermediate results and then using another MSJ operator to join it with the remaining relations. Doing so gives the intermediate result its own store and any new incoming tuples of the other relations can simply probe this intermediate result store instead of the individual relation stores. This has been shown to improve performance as it significantly decreases the probe count.

The late materialization model introduced in this work is intended to be implementable in systems like CLASH. We adapt several concepts used in CLASH for computing streaming joins in order to conceptualize the late materialization model. These include:

1. **Relation stores:** Each relation has its own store within the system for storing incoming tuples persistently. New tuples are first forwarded to corresponding stores and saved. These will also be used to probe other relation or intermediate result stores to find potential join partners.

2. **Intermediate result stores:** Depending on the structure of the join

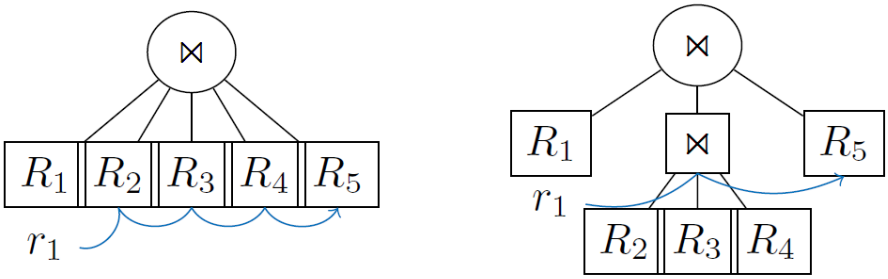


Figure 2.2: Linear MSJ vs. MSJ tree, Source: [16]

plan, the required intermediate join results are also saved in their corresponding stores.

3. **Tuple routing mechanism:** Optimized routing of input tuples such that leaf level joins that need to be materialized can be computed directly in the relation stores. This mechanism also dictates how the materialized results will be routed to the remaining relation stores and viceversa to guarantee that all results are observed.

Since the scope of this work is to create and test a late materialization model, many of the more advanced features in CLASH are not adapted. Some examples would be automatic Storm topology generation or join plan optimization. We therefore restrict our experiments to manually written topologies for different join plans in order to evaluate the effectiveness of late materialization. Deciding between late and early materialization is also an important factor since late materialization is not always beneficial [12]. Since CLASH expects selectivities and tuple arrival rates as meta data from the user, we will define some thresholds based on these values using which it will be possible to decide between the two approaches.

2.3 Materialization Options

In traditional row oriented stores, result materialization of join queries is not an obvious concern. This is mainly because several powerful and widely accepted join computation algorithms already exist. Rastogi [20] explains the methods for join processing available in PostgreSQL namely nested loop join, hash join and sort-merge join. He also presents some sample scenarios where each algorithm is chosen by the optimizer to run the join queries. These existing join algorithms are already powerful enough and the need for late materialization of results is not very prevalent in row stores.

Column stores have shown to perform comparatively well compared to row stores in terms of analytic queries and decision support applications. This has resulted in a noticeable amount of research work on investigating the effects of varying join result materialization methods on column oriented stores [11, 12]. It makes more sense in this case as well because for column oriented stores there

is an additional overhead of constructing tuples before or while working with them. Since each column of a particular relation is stored in separate locations on disk, to get a complete tuple all of its columns need to be fetched separately and then combined. It is possible to do this in many different points within the query plan and it is not always clear when the optimum time for constructing the full tuples is going to be.

Early Materialization Results of a query are said to be materialized early when intermediate result tuples are fully constructed as soon as it is possible to do so. This can happen during very early stages of the query plan. This can be illustrated by using an example with a column oriented store. Given a relation R with columns $R.a$, $R.b$ and $R.c$ and a query with selection predicates on all three of R's columns. When using an early materialized approach, the query processor would first fetch and stitch together the columns for a block of tuples of R and then apply the filtering criteria and let the tuples which satisfy it to pass through. This can sometimes be inefficient especially when selectivity is low. For cases where the query selectivity is low, the CPU will spend a lot of resources fetching blocks from the disk and stitching them together but in the end most of these will fail to pass the selection criteria. On the other hand, in case of high selectivity, this approach is very much suitable. This is true because in case of high selectivity, there will be a very large number of join results. The overhead of reconstructing these later will be much higher compared to working with already constructed tuples to begin with.

Late Materialization Results of a query are said to be late materialized when no intermediate results are fully constructed. Intermediate results or data needed to continue query processing will only contain the minimum required information, i.e., information that is needed to complete the join operations. These include primary key and foreign keys and also any other attributes which are involved in the join operation such as a filtering criteria or something similar. At the very end of the query plan the result will usually consist of a set of identifier values which can be used to recover the appropriate tuples from the relation stores and perform reconstruction.

Late materialization of join results is very common for column oriented stores [11, 12]. The storage mechanism in column stores is generally abstracted from the user. The user will always work with full rows just like in a regular row store. In the background the query processor takes care of fetching and combining column data for individual rows. Since tuple construction is such an inherent feature in column oriented stores, taking advantage of late materialization is very sensible. But even then it is not always obvious whether or not the result materialization should be delayed. The cost of reconstructing tuples at the end of the query should not be higher than what it would have been with eager materialization. Application of late materialization of join results is not commonly seen in row stores. Data streams are also row oriented in nature as each stream is defined by a set of field names and data types which is analogous to column headers. But there are options of applying late materialization in stream join processing frameworks in order to increase performance.

Looking back at the column store example mentioned above, a similar query with late materialized results will be processed differently. Instead of fetching and then combining the individual column data for a block of tuples, for the late materialized approach the idea would be to first apply the selection criteria on each of the columns separately and then obtain position lists of the tuples that satisfy the criteria. The position list could simply be a bit vector of size equal to the number of tuples in the relation. A value of 1 in a cell would then indicate that the row satisfies the selection predicate. In this manner three position lists will be obtained for the three columns $R.a$, $R.b$ and $R.c$. Now a bit wise AND operation can be performed between these position lists to obtain a final list that satisfies all three filtering criteria. This is essentially the end of the actual query plan. Now the final position list must be used to re-access the tuples which are valid and get the actual result. Accessing tuples for each of the columns and then combining these into full rows is a CPU intensive process. By using late materialization we omit this entirely. But the reconstruction process in this case introduces a new overhead.

Chapter 3

Related Work

Stream join processing is different than conventional join processing. This is primarily because the nature of data streams is different than traditional relational data. Streams are continuous and results are expected to be continuous as well. Complications also arise when considering whether or not the query processing is done over bounded windows or over the full history. We will discuss contemporary research that deals with designing efficient and correct stream join processing algorithms and take into consideration the above mentioned issues. We will also discuss recent progress in application of late materialization in join processing for various platforms.

3.1 Streaming Join Implementation Approaches

When dealing with data streams, it is not possible to have the whole data set available at a given point in time. This rules out several standard join algorithms which use blocking operations like sort merge join. Also it is necessary to retain all observed tuples in the streams because any new incoming tuple might be a join partner of an arbitrary previous one. This becomes a huge problem for storage and also retrieval when there is an unbound continuous streaming source. An example would be data streaming in from sensor networks. Xie et al. have summarized several approaches to solving such stream join processing problems [13]. A common solution is the sliding window approach. This restricts the join computation to the tuples present within a fixed window. This window can be defined by a time interval or number of tuples observed. This solves the problem of unbounded tuples but on the other hand creates a possibility missing some results because a tuple will never find join partners which exist outside its own window. But since the window is defined by the user, any results that are missed because of the window bounds are not important. Generally proposed stream join algorithms utilize some sort of hash join, e.g., XJoin [24]. These algorithms can also dump some parts of the hash table to the disk and process these later. This does not maintain the order of the results but it is possible to re-sort the results based on timestamp. This is one way in which limited memory can still serve an unbounded stream while guaranteeing results. There is also a possibility of using statistics to try and determine exactly which tuples to keep in memory and which to spill out into the disk. The RPJ [22] algorithm uses probabilities to determine which tuples are more likely to join next and keeps those in memory while dumping the rest.

When constructing a model for late materialization these problems must also be considered. When results are fully materialized at a much later stage, it is important to ensure that the tuples that are a part of this result are still available in the stores. So older tuples need to be kept in memory longer to ensure that a future re-construction request can be fulfilled. But then again we are always

limited by how long tuples can be kept in store. If a disk dump approach is present, to complete reconstruction, first checking the main memory stores is needed to see if the relevant tuple is found, if not then the disk store can be checked. But this introduces even more overhead to the reconstruction process. There are several methods by which the stream joins can be optimized. But these are more relevant to the join processing system in consideration. The late materialization model will only be affected by the duration for which tuples are available. As long as tuples are not completely lost, reconstruction is possible and therefore late materialization of results will work.

The AMJoin algorithm presented by Kwon et al. [18] is an algorithm for joining multiple data streams which can detect join failures in constant time. When joining multiple relations, a join failure is observed when any one of the available joining conditions fails. The idea is that for a join query containing many such conditions, failure of one condition is enough to ignore the rest. To achieve this, the authors propose a new hybrid data structure called Bit-vector Hash Table [18]. The AMJoin algorithm improves over the previously defined MJoin algorithm by Viglas et al. by avoiding unnecessary probing [25]. The MJoin algorithm maintains a hash table for each input stream. The range of hash values of the join keys in the input stream defines the address space of the hash table. For each incoming tuple the hash value of the join key is calculated and the corresponding bucket in the appropriate hash table is used to store it. Next this hash value is used to probe the other hash tables. If this hash value is not found in any of the other hash tables the join fails. In MJoin it takes *(n-1)/2* probes to detect that a newly arrived tuple does not have any join partner. Hence there is potential to avoid this and improve performance. AMJoin does this by using a bit vector hash table. The address space of this table contains hash values of the keys observed till current time point. Each key also contains a corresponding bit vector of length equal to the number of streams involved in the join plan. When a new tuple arrives for input stream n, hash value v of its join key is calculated and the corresponding bit vector in the bit vector hash table is updated by setting the *nth* bit to 1. Once this is done, the entire bit vector of v is checked. If all bits are not set to 1 then the join fails immediately. Experimental evidence was also presented to show that AMJoin join is able to outperform the original MJoin algorithm.

The *join-biclique* stream joining model and a scalable distributed stream joining system *BiStream* based on the mentioned model have been presented by Lin et al. [19]. The model works by separating available processing units into two distinct groups. And then creates a bipartite graph between them. Given there are two relations (which are the input streams), each relation will be stored in only one side of the bipartite graph. Therefore the edges connecting two sides of the bipartite graph represent potential join result. As it is possible to get a Cartesian product of the two sides of the bipartite graph, computation of all join predicates is possible in this model. The BiStream system is also based on this model. It consists of two layers, the router layer and the joiner layer. In the router layer, all input streams enter through the shuffler and are randomly sent to dispatchers which decide which side of the joiner to send it to. The joiner layer performs the actual join operation. The authors have discussed in great detail how load balancing takes place and also how the join protocols are designed so that none or very few join results are actually missed. Also

experimental evidence has been provided to show that the model and system works well in distributed environments and is also scalable.

3.2 Current Applications of Late Materialization

Contemporary work on application of late materialization in join query processing [12, 15] revolves around relational column oriented stores with most simply commenting on its applicability to row stores as well. Late materialization goes very much hand in hand for column oriented query processing [12]. This is because the input columns stored on disk must eventually be converted to rows. But at what point in the query plan this should be done is not always obvious. In this case a late materialized approach would first apply the conditions on each column and filter them out and obtain position lists of satisfying rows, then perform a position wise AND operation to intersect these position lists. Finally re-access the columns in order to reconstruct full tuples. This shows potential to be more CPU efficient as fewer intermediate tuples need to be combined. But the re-scanning process to complete reconstruction can also be slow so it becomes an overhead. It has been showed that in certain cases better performance is obtained when using late materialization compared to early materialization.

We encounter the same problem when applying late materialization to stream joins. Instead of transferring the whole tuple within stores during join computation we transfer only those attributes that are necessary to compute the full join results. This is efficient because the amount of data transferred is going to be less. But during reconstruction, it is required to probe each of the stores again (the order does not matter since we are only reconstructing tuples) and this becomes slow especially because the number of tuples keeps growing with time causing retrieval to also become slower. This very important point is made by Abadi et al. where it is mentioned that even though late materialization can have several benefits like less storage requirements, faster result processing and fewer unnecessary intermediate results, if the data re-access cost is high then there is too much overhead [12]. In our model, we have tried to define some quantification as to exactly in which scenario late materialization of results may provide performance benefits. It depends on several factors such as number of relations involved in the join, size of the join attributes in comparison to the non-join attributes and also the overall selectivity.

Abadi et al. [11] have introduced a concept called Invisible Join which utilizes late materialization for computing analytic queries in data warehouses which have star schema style tables and work with foreign-key/primary-key joins. Analytic queries for data warehouses usually involve several join conditions followed by some form of grouping or ordering or both. The join conditions within the query also include predicates for the dimension tables. The authors propose that by treating the join conditions for dimension tables as separate selection queries, it is possible to run several of these in parallel. So for each join condition involving a dimension table, a selection query is performed on the corresponding

table. Then a set of foreign keys which satisfy the given condition is obtained. For example if there is a *customer* fact table and a *region* dimension table, a join condition such as *customer.region = 'ASIA'* can be translated to a selection query in the region table itself. Then the customer keys of the satisfying tuples are obtained and stored in a hash table. In the next stage the values from these hash tables can be used to probe the fact tables corresponding key columns in order to obtain *position lists*. These are positions of the tuples which satisfy the previously computed predicates. Now a bit wise AND operation can be performed between these position lists to obtain the final position list which satisfies all the join predicates. Now this final position list can be used to construct the full result including required information from the dimension tables. It is also seen that the reconstruction overhead mentioned by Abadi et al. [12] does not occur here because dimension tables are usually very small and almost always contain foreign key columns. Hence for this specific scenario late materialization proves to be fruitful.

Deo [15] worked on the application of late materialization in the traditional sort merge join algorithm. The authors describe the possibility of having partial late materialization, where full tuples are formed only after individual predicates are applied. This is still partial because the tuples that are intermediately constructed may not satisfy the other join conditions within the query. While in case of full late materialization the tuple reconstruction will wait until all the predicates in the join plan are processed. In our late materialization model we have gone with full late materialization. The reconstruction starts only when all the join predicates have been evaluated for an incoming tuple.

Chapter 4

Approach

We have discussed earlier in Chapter 2 certain implications which do not allow traditional join algorithms to be used with streaming data. Streaming data is generally treated and processed in a row oriented manner. This can be seen in Apache Storm's [4] core abstraction of *streams* where a stream is defined by a set of fields. These fields are analogous to columns in a row store. Each incoming tuple is then mapped to the stream by assigning the field values which makes it a row. So typical approaches used for late materialization in column oriented scenarios is not applicable here.

When working with data streams and Storm topologies a lot of inter topology communication takes place. Tuples are generally processed partially in one bolt and then passed on to the next and so on until the processing is finally complete. So, for example, we have an intermediate join result that we would like to compute and store in one of the bolts. This join result would consist of all the fields from both the relations it was computed from. It can be noticed that this is not always necessary. Since this is an intermediate result, there is always a possibility that this may not eventually materialize into a final join result. So instead of emitting the entire tuple with all its fields, it is possible to only emit the identifier fields and any other fields which are needed further in the joining process. This poses several advantages such as reduced network traffic and faster processing within the bolts which will allow them to accept incoming tuples at a much higher arrival rate i.e. lower chances of getting bottle necked.

By using such a late materialized approach the final result emitted from the topology will simply be a set of identifiers. Each tuple in the result will only contain values pointing to actual tuples in the relation stores. So in a fashion similar to that of column stores, a reconstruction phase must now be completed in order to obtain the actual results. The reconstruction can be done in different ways:

- After the final set of identifiers is ready, probe each of the relation stores one by one and during each probe fetch and attach the tuple from that store. This is the most straight forward approach.
- Decoupling the reconstruction process entirely. A separate process would handle the reconstruction by taking the set of identifiers as input and then probing the stores to obtain fully constructed tuples.

It is important to note that reconstruction is an expensive operation and can effect overall processing speed of the topology if not done efficiently. In fact it might sometimes be a deciding factor on whether or not to use late materialization for a particular operation. How factors such as selectivity play a big role in reconstruction overhead will be discussed in detail later in the chapter.

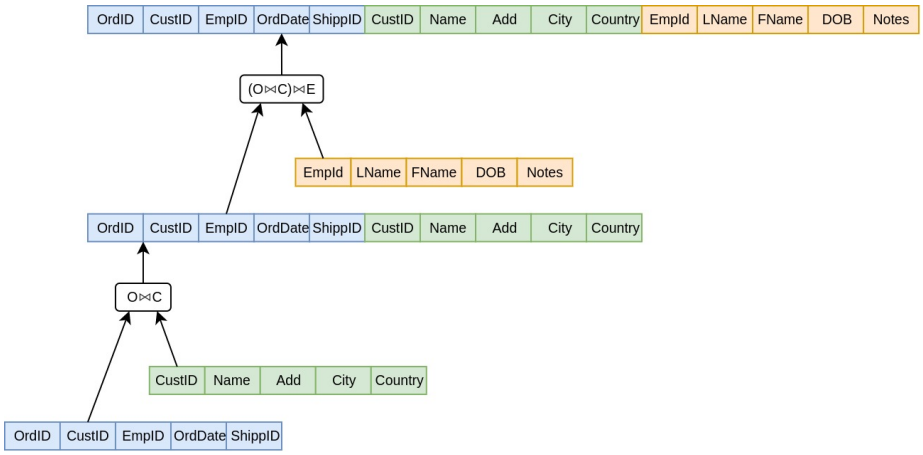


Figure 4.1: Join computation using early materialization

4.1 Late Materialization Model

In contrast to early materialization, where during join computation all the columns of the relations involved in the join are maintained and moved around, late materialization of results involves only transferring the join keys during the process of finding join partners. So once all parts of a join tree are fully computed, instead of having the result tuples with all columns already there, for a late materialized result only the join keys will be present. These keys can then be used to retrieve the tuples from the individual relations and then reconstruct the full result tuple. The examples in Figure 4.1 and 4.2 show three relations involved in an equi-join and how the processing will vary for late and early materialization.

In Figures 4.1 and 4.2 the early and late materialization are clearly distinguished. Even though the above examples have been provided for three relations, it is possible to extend this to n relations, where each time a new relation is joined with the previous results, only the **join keys** are being transferred. Even though the advantages of having late materialization are not very clear when considering regular relational databases, for join computation in distributed stream processing it will play a key role,

- Reduced network traffic when working with large tuples or several streaming sources
- Faster processing times within network resulting in fewer incoming tuples being discarded due to busy working nodes.

4.2 Base Topology

When dealing with join operations in data streams we must keep in mind that two data streams operating independently will have tuples arriving at different

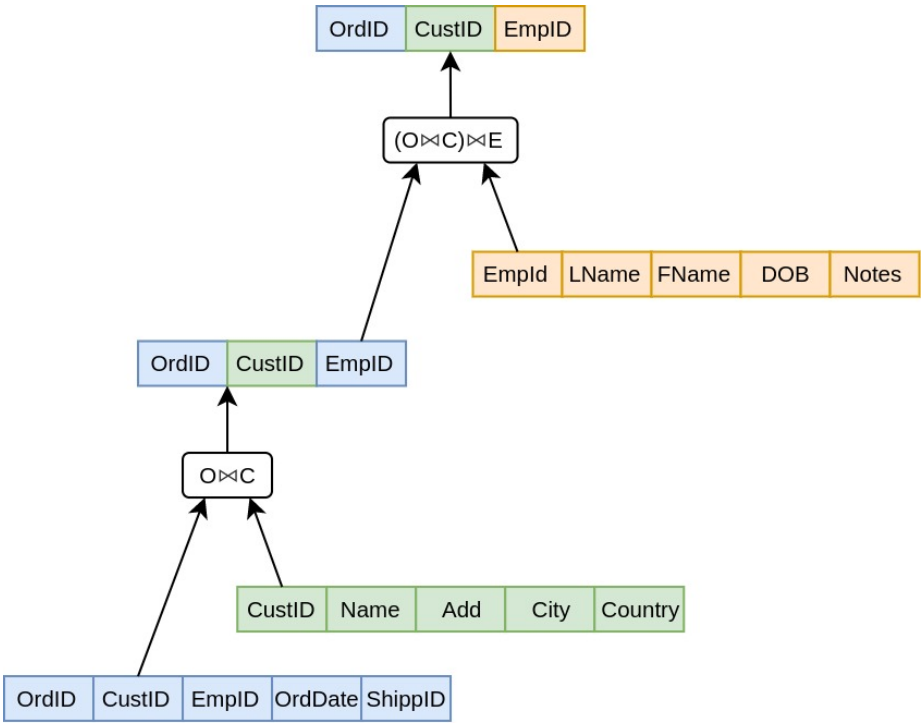


Figure 4.2: Join computation using late materialization

times. Given a tuple of relation R1 arrives at time T1, and a tuple of relation R2 arrives at time T2 where T2≫T1, if these two tuples form a join result then this result may never be observed. It is necessary for the R2 tuple to arrive within the time frame for which the R1 tuple is retained within the system. If not, then the R2 tuple will not find the potential join partner in R1. This problem can be solved by using temporary stores for each of the relations involved in the plan. In this way each tuple arriving will go to its own store and be kept there for some time. This will allow potential join partners arriving at a later time to still find a join result. Longer storage duration for tuples will mean that fewer join results will be missed. But there is always a trade off between storage duration and space requirements.

The use of stores is not limited to the individual relations, stores can also be used to materialize intermediate join results. In Figure 4.3 a join plan between relations R1, R2 and R3 can be seen. The green coded join symbol refers to that particular join result being materialized. This intermediate result R1⋈R2 now has its own store. Incoming tuples for relation R3 can now simply probe the R1⋈R2 store to compute the final result.

Given three streaming relations R1, R2, and R3 and the join plan from Figure 4.3 we can define the following operations:

For incoming tuples of relation R1:

- Upon arrival, a R1 tuple will be saved in the R1-store and forwarded to the R2-store for probing.

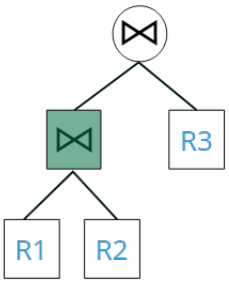


Figure 4.3: $R1 \bowtie R2$ materialized

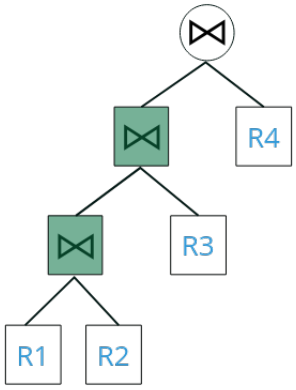


Figure 4.4: $(R1 \bowtie R2) \bowtie R3$ materialized

- If the probe is successful, the R1⋈R2 result is saved in the R1⋈R2-store and forwarded to the R3-store for probing
- If this probe is successful the final result (R1⋈R2)⋈R3 is emitted directly from R3-store.

For incoming tuples of relation R2:

- Upon arrival, a R2 tuple will be saved in the R2-store and forwarded to the R1-store for probing.
- If the probe is successful, the R1⋈R2 result is forwarded the R1⋈R2-store and also the R3-store for probing.
- If this probe is successful the final result (R1⋈R2)⋈R3 is emitted directly from R3-store.

For incoming tuples of relation R3:

- Upon arrival, a R3 tuple will be saved in the R3-store and forwarded to the R1⋈R2-store for probing.
- If this probe is successful the final result (R1⋈R2)⋈R3 is emitted directly from the R1⋈R2-store.

The above set of operations can now be modeled as a Storm topology (Figure 4.5). The topology can be used to find the join result between R1, R2, and R3 with the (R1⋈R2) intermediate result being materialized.

4.3 Cost Model Fundamentals

In order to compare late and early materialization we will now define a standard cost model which can be used to calculate the storage requirements and overall

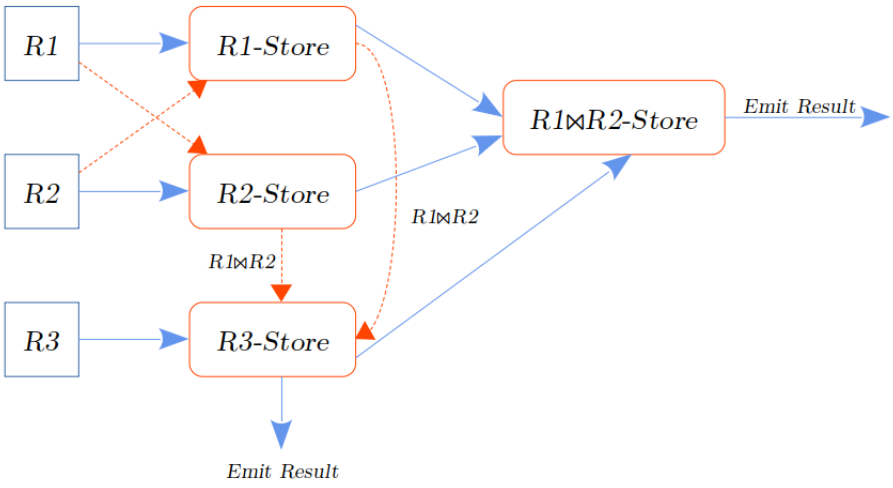


Figure 4.5: Base Storm topology

communication costs that are incurred when computing joins over the base topology. As mentioned before the core idea behind late materialization of join results is to to transfer only the minimum amount of information within nodes in the topology while the join computation is ongoing. This minimum amount consists only of the join attributes of the relations involved, leaving the non-join attributes behind. The set of non-join attributes of a relation is its **payload**. We define this formally as follows,

Given a set of schemas $R_1, ..., R_n$, each schema is separated into two groups, one for the join attributes and one for the non-join attributes.

$$K_n \text{ is the set of join attributes of schema } R_n$$

$$P_n \text{ is the set of non-join attributes of schema } R_n$$

We now define the functions $W_k(R_n)$ which is the total size (henceforth in bytes) of the join attributes of a single tuple of the schema R_n and $W_p(R_n)$ which is the total size in kilo bytes of the payload of a single tuple of the schema R_n.

It is assumed that each schema in consideration contains a small number of join attributes and a large number of payload attributes. Meaning that,

$$W_p(R_n) \gg W_k(R_n)$$

The size of a full single tuple from any schema R_n is therefore given by,

$$W_k(R_n) + W_p(R_n)$$

Given a tuple from schema R_n and a tuple from relation R_m form a join result or intermediate join. The size of this is denoted as follows,

$$W_k(R_n \bowtie R_m) + W_p(R_n \bowtie R_m) = W_k(R_n) + W_p(R_n) + W_k(R_m) + W_p(R_m)$$

If only the size of either the join attributes or the payload is required, only the required function is selected. The above definitions are on a per tuple basis. However, when dealing with streams we do not have a fixed number of tuples

for a given schema, as new tuples are always coming in. Therefore we must compute storage and network traffic costs incurred during join computation up until a given time T. We can now define the following:

Total number of tuples of schema R_n observed at time T: $|R_n|$
Selectivity between two relations R_n and R_m: $f_{n,m}$

Using the above functions and notations we will, in a later section, develop generalized cost models for storage and communication for both late and early materialized approaches and assess the effectiveness of late materialization.

4.4 Early Materialization

For the early materialized approach using our base topology, we will always transfer the full tuple (keys+payload) from node to node whenever required. An example of this would be when the intermediate join $R1 \bowtie R2$ is computed in either R1 or R2 store, join attributes and payloads from both tuples are transferred and stored in the $R1 \bowtie R2$ store. From where it is further transferred to the R3 store and so on. An advantage of computing the joins like this is that the full result is readily available at either the $R1 \bowtie R2$ store or R3-store and can be directly emitted. But it should be noted that since the intermediate join result is being stored and communicated along with the payloads, this will generate a large amount of traffic within the topology.

Cost Model for Base Topology Given three relations R, S, T and the base topology defined before, we now define the storage and communication requirements when using the early materialization approach. The base toplogy uses a join plan with the $R \bowtie S$ intermediate join materialized. So we will have a separate $R \bowtie S$ store for storing these intermediate results.

Given the selectivities and total number of tuples observed till time point t, the number of intermediate join results of R and S relations is given by P,

$$P = f_{R,S} * |R| * |S|$$

Therefore, the storage costs are given by:

- **R-Store** cost $= |R| * \{W_k(R) + W_p(R)\}$
- **S-Store** cost $= |S| * \{W_k(S) + W_p(S)\}$
- **T-Store** cost $= |T| * \{W_k(T) + W_p(T)\}$
- **(R $\bowtie$ S)-Store** cost $= P * \{W_k(R \bowtie S) + W_p(R \bowtie S)\}$

Total storage cost is then the sum of the above costs. We can determine the total amount of communication within the topology by observing the which routes tuples take. Using the base topology will give the following communication costs,

1. $R \rightarrow S$ cost $= |R| * \{W_k(R) + W_p(R)\}$
2. $S \rightarrow R$ cost $= |S| * \{W_k(S) + W_p(S)\}$

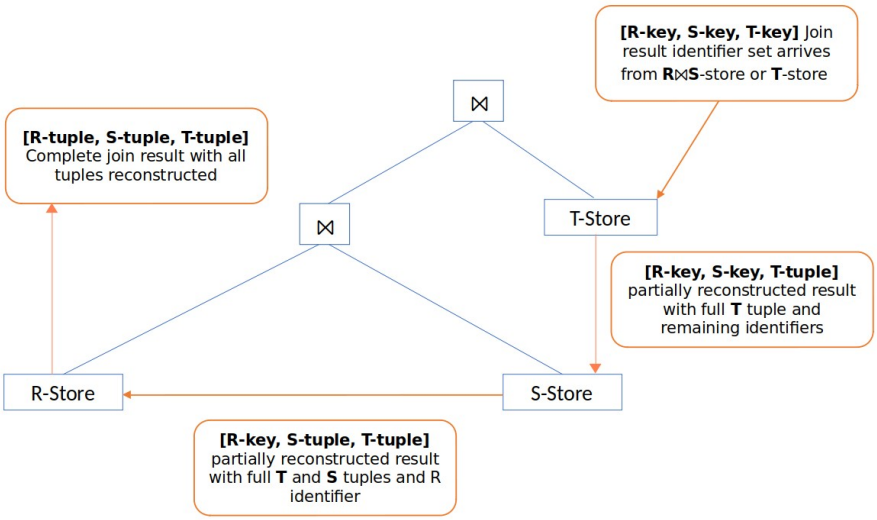


Figure 4.6: Tuple reconstruction workflow

3. $R \rhd\lhd S \rightarrow T$ cost $= P * \{W_k(R \rhd\lhd S) + W_p(R \rhd\lhd S)\}$
4. $T \rightarrow R \rhd\lhd S$ cost $= |T| * \{W_k(T) + W_p(T)\}$

We can see from the above equations that the storage and communication costs are equal when using the early materialized approach. This is true because when the join result is eagerly materialized it is required to transfer full tuples along the topology at all times. When obtaining similar equations for late materialization, we will see there is a decrease in the amount of information that is both stored and communicated.

4.5 Late Materialization

In contrast to early materialization, here we aim to minimize all forms of intermediate storage and communication that is encountered during the join computation process. The core idea is to only traverse the join-attributes over the network whenever possible. Once the whole join plan is complete, the result will be a set of tuples consisting only of the join-attributes from each of the relations within the plan. This approach is based on the invisible join concept introduced by Abadi et al. [11]. The approach there was to use a position list to store the position of the tuples of a relation which satisfies a given selection predicate. Then a bit-wise AND operation was performed between the position lists. The resulting final list was used to extract tuples individually from each corresponding relation. This approach can be modified to be applicable in computing streaming joins as well.

In order to emulate the invisible join concept, we design the topology in such a way so that all the relation and intermediate join stores always emit join-

attributes only. This could be just the primary key column, multiple foreign key columns or any chosen join-attribute of the relation. The intermediate join stores will then store only join-attributes for the involved relations instead of the whole tuple. Finally, when the join plan is complete, these join-attributes can be used to reconstruct the tuples from each relation within the result. The reconstruction is done by traversing through each individual stores and using the respective join-attributes to find the proper tuple. Once the tuple is found it is fetched and attached to the result, which is then forwarded on to the next relation store and so on. The following example illustrates how the reconstruction process takes place.

In Figure 4.6, we see the work flow for reconstruction of a join result between three relations R, S, and T, $(R \bowtie S) \bowtie T$, computed over the base topology. The initial result of the join computation is a set of keys. These are the identifiers of the actual tuples in the stores. The set **[R-key, S-key, T-key]** is generated on either the T-store or the $(R \bowtie S)$ store depending on what arrived first. The goal is now to retrieve the corresponding tuples for each of the keys and then get the actual result.

1. The T tuple is reconstructed first. This is convenient when the join result is generated in T store itself. In case if it is generated in $(R \bowtie S)$ store then this must be transferred to the T store to start reconstruction. It is not mandatory to start reconstruction from T store in fact we can start from whichever store we want. Once the appropriate payload is added the result is then forwarded on to the S store along identifiers for relations R and S.

2. At the S store, the correct payload is fetched and attached to the result. This is then forwarded to the R store along with the last remaining identifier for relation R.

3. Finally at the R store, the required payload is fetched and attached to the result. This makes the result complete and can now be emitted out of the topology.

The above process is applicable for all tuples that form join results. An important point to note here is that the reconstruction order is not fixed so there is an option to optimize this. The optimization approach presented by Abadi et al. [11] uses the selectivity values of the predicates to determine which one to execute first. The order goes from most selective predicate to the least selective one. A similar approach can be adapted here in the tuple reconstruction process. We have introduced the join-attribute and payload concept earlier. Since reconstruction involves adding a full tuple to a result and then transferring it to another store and repeating this until all stores are traversed, it makes sense to reconstruct the lightest tuple first. That way the heaviest tuples travel through the topology the least. This can be done by performing the reconstruction in ascending order of tuple payloads.

Since tuple payload sizes are not available readily, one possible way to determine the reconstruction order is to maintain the average payload size for each relation in a buffer. The average payload will obviously be for tuples observed till a given point in time. This can then be used to get a proper reconstruction order. This

payload buffer can be updated at certain intervals to ensure that the values are as recent as possible.

Using late materialization we are able to save storage costs overall and also communication costs during join computation. Once the join result in the form of a identifier list is obtained, the reconstruction process has to be initiated. Compared to early materialization, this is an overhead. It should also be noted that tuple reconstruction involves a fair amount of communication as the result is partially constructed and moved on from store to store. This motivates the need to determine a threshold, based on certain parameters like predicate selectivity or tuple count, above which late materialization does not yield any benefits. This has been further elaborated in the Effectiveness of late materialization section.

Cost Model for Base Topology The model for computing communication costs for the late materialized approach has been divided into two parts, cost of join computation and cost of reconstruction. This is because when results are lazily materialized there are two distinct phases that take place. The first phase involves actually finding the join results. This process is similar for both approaches. But at the end of this phase, for late materialization, we only get an identifier list, not the whole result. Therefore another phase of tuple reconstruction is executed in order to retrieve the appropriate tuples from each individual store and obtain the full result. We will elaborate communication cost model for late materialization based on this concept.

Given three relations R, S, T and the base topology defined before, we now define the storage and communication requirements when using the late materialization approach.The base toplogy uses a join plan with the $R \bowtie S$ intermediate join materialized. So we will have a separate $R \bowtie S$ store for storing these intermediate results.

Given the selectivities and total number of tuples observed till time point t, the number of intermediate join results of R and S relations is given by P,

$$P = f_{R,S} * |R| * |S|$$

Therefore, the storage costs are given by:

- **R-Store** cost $= |R| * \{W_k(R) + W_p(R)\}$
- **S-Store** cost $= |S| * \{W_k(S) + W_p(S)\}$
- **T-Store** cost $= |T| * \{W_k(T) + W_p(T)\}$
- **(R $\bowtie$ S)-Store** cost $= P * \{W_k(R \bowtie S)\}$

Total storage cost is then the sum of the above costs. We can determine the total amount of communication within the topology by observing the which routes tuples take. The join computation phase will have routes similar to the early materialized model. For the reconstruction phase the routes will follow the tuple reconstruction method shown in Figure 4.6. The idea is to start the reconstruction at any one of the stores, preferably the one with the lowest observed average payload. And then move from store to store in ascending order of average payload. Initially we start with a set of tuple identifiers. Once a tuple

is fetched and attached to the result, this along with the remaining identifiers is forwarded to the next store and so on until all stores are traversed.

Communication costs for join computation:

1. $R \rightarrow S$ cost $= |R| * \{W_k(R)\}$
2. $S \rightarrow R$ cost $= |S| * \{W_k(S)\}$
3. $R \bowtie S \rightarrow T$ cost $= P * \{W_k(R \bowtie S)\}$
4. $T \rightarrow R \bowtie S$ cost $= |T| * \{W_k(T)\}$

Communication costs for tuple reconstruction: We assume the following reconstruction order in this example, $T \rightarrow S \rightarrow R$. Since all the join results will eventually be reconstructed, we enumerate the total number of join results from the query using the parameter Q:

$$Q = f_{R,S} * f_{R,T} * f_{S,T} * |R| * |S| * |T|$$

The reconstruction begins at T store, where we fetch the payload for relation T and then move on S followed by R fetching corresponding payloads as we go. The fully constructed join result can be directly observed from the R store. The costs incurred for communication during reconstruction are as follows:

1. $T \rightarrow S$ cost $= Q * \{W_k(T) + W_p(T) + W_k(S) + W_k(R)\}$
2. $S \rightarrow R$ cost $= Q * \{W_k(T) + W_p(T) + W_k(S) + W_p(S) + W_k(R)\}$

In general we always have $n - 1$ reconstructions for all n relations involved in the join plan. This is because at the final store, the join result can be directly emitted and no further communication within the topology is needed. The first step of reconstruction completes the T tuple and forwards it along with the S, R-keys to the S-store. Here the S tuple is added and forwarded to R-store where the final R tuple is added and the result is emitted.

Using the late materialized approach we are able to save storage requirements for intermediate join results. Intermediate results are not always guaranteed to be final results eventually. So, by only storing the join-attributes for intermediate results, the cost of saving and communicating unnecessary intermediates is minimized. This is especially true when $W_p(R) \gg W_k(R)$. We also notice that communications for join computations only involve join-attributes and none of the payloads. This will allow the topology to operate efficiently even when the streaming source is generating tuples at a very high rate or the data volume of the tuples is very large (high payload).

The reconstruction phase on the other hand, as mentioned before, will at some point become an overhead. This maybe when the join selectivity is too high. Or when the number of relations involved in the join plan are very large. Then the reconstruction phase will require more time to complete. It is possible to decouple the reconstruction phase and the join computation phase among two separate systems. While one topology and processes the join the other, asynchronously reconstructs result tuples. Even though the final results may appear in a slightly delayed fashion, the join computation topology will be able to process incoming tuples much faster since it does not have to worry about the reconstruction phase. In cases were such an optimization is not possible we

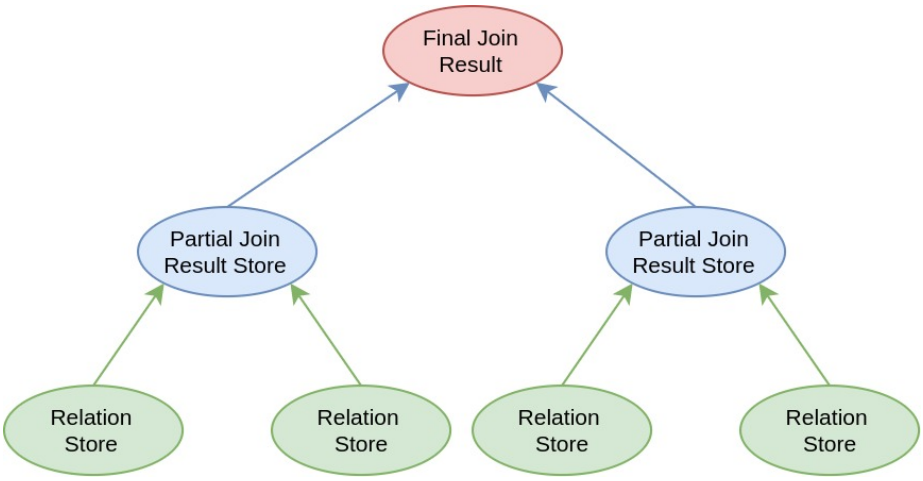


Figure 4.7: Join plan represented by stores as nodes

can still have a threshold predefined as mentioned before. This can be used to determine in which scenarios late materialization will actually be useful.

4.6 Generalized Cost Models

The cost models that have been defined before are specific to the base topology that we introduced earlier. Here we will define a more general model which can be implemented in any stream join processing framework that utilizes the concept of intermediate result stores. We assume that the join processing framework will provide the result materialization model with an optimum join plan. We model the given join plan as a tree where leaf level nodes represent relation stores and inner nodes represent intermediate join result stores. For now we will define general cost models for binary join plans only. Later on we discuss the complexities that arise when computing costs, especially communication costs, with multi way joins. The cost models are constructed using recursive formulas based on the example tree structure representation in Figure 4.7. The costs are calculated in a bottom up fashion. This means that computing the node cost at the root level node where final results are obtained will give the cost of the entire plan. The unit of costs depends on the unit being used to measure $W_k(R)$ and $W_p(R)$. If we measure payload sizes in the bytes then our overall costs will also be in bytes.

Early Materialization Given a join operation involving n relations $(R_1, ..., R_n)$ and an optimized join plan similar to Figure 4.7, $S_{cost-early}(m)$ in Equation 4.1a can be used to compute the storage costs of a given node m when using eager the materialization approach. We define to distinct cases for m in our equation. If m is a leaf node, then it is a relation store. So the storage cost is simply the total size of all the tuples present in m. We have defined this in the same manner as our example cost models before where $W_k(R)$ is the set of join at-

tributes for a single tuple of the relation R and $W_p(R)$ consists of the additional columns/payload of a single tuple of R. Since for our model we are considering binary join trees, if m is an inner node then its storage cost is given by $F(m)$ plus the storage costs of it's child nodes. This allows for the bottom up cost computation mentioned before. $F(m)$, defined in Equation 4.1b, is the cost of storing the partial join results between a relations $(R_1, ..., R_a)$ in the node m. When considering binary joins, any inner node m would therefore only contain two such relations. In $F(m)$ the expression: $\sum_{i=1}^{a} (W_k(R_i) + W_p(R_i))$ is the distinctive feature of early materialization. Interim join stores will store both join attributes and payload information allowing results to be emitted immediately. We will see later on how this changes when the late materialized approach is used.

An interesting juncture here is how we actually define $W_k(R)$. This set of join attributes actually contains two types of data, one being the join attributes themselves which are needed for further join computation and the other being the identifier information for the relation (primary key or unique ID). When considering foreign key joins, like we do in our experiments, the join attributes are always identifiers as well. So no clear distinction is made in this case. But when we look at non foreign key joins, it is possible that the join attributes are not the same as the tuple identifiers. In this case we can remove a join attribute from the $W_k(R_i)$ set after it's operation has been performed since this attribute will not be needed anymore.

$$S_{cost-early}(m) = \begin{cases} |R| * \{W_k(R) + W_p(R)\} & \text{if m} \in \text{leaf nodes} \\ F(m) + S_{cost}(\text{m.left}) + S_{cost}(\text{m.right}) & \text{if m} \in \text{inner nodes} \end{cases} \tag{4.1a}$$

$$F(m) = \prod_{\substack{i=1 \\ j=1}}^{\substack{i=a-1 \\ j=a-1}} f_{i,j} * \prod_{1 \leqslant i \leqslant a} |R_i| * \sum_{i=1}^{a} (W_k(R_i) + W_p(R_i)) \tag{4.1b}$$

Equations 4.2a can be used to compute the communication costs $C_{cost-early}(m)$ incurred at a node m for a given join plan when using the eager materialization approach. Since the goal here is to materialize results as soon as possible, all tuple information is transferred whenever it is required. This is given by $\sum_{i=1}^{a} (W_k(R_i) + W_p(R_i))$ in Equation 4.2b. Meaning that the intermediate result stores will transfer join attributes and payload information for all the relations which are part of it's results. On the contrary, the late materialized approach would try to minimize this by only transferring what is necessary for further join computation in order to reduce total communication cost.

$$C_{cost-early}(m) = \begin{cases} |R| * \{W_k(R) + W_p(R)\} & \text{if m} \in \text{leaf nodes} \\ G(m) + C_{cost}(\text{m.left}) + C_{cost}(\text{m.right}) & \text{if m} \in \text{inner nodes} \end{cases} \tag{4.2a}$$

$$G(m) = \prod_{\substack{i=1 \\ j=1}}^{\substack{i=a-1 \\ j=a-1}} f_{i,j} * \prod_{1 \leqslant i \leqslant a} |R_i| * \sum_{i=1}^{a} (W_k(R_i) + W_p(R_i)) \tag{4.2b}$$

Late Materialization Using the same scenario as before we now define the storage cost for a given node m when using the late materialized approach as $S_{cost-late}(m)$ in Equation 4.3a. The cost for leaf nodes or relation stores is the same as before. We do not have an option of skipping anything here because these stores are responsible for storing full incoming tuples so that later during reconstruction these can be properly accessed again. Primary difference with the early materialized model can be noticed in Equation 4.3b. We define $F(m)$ such that it only stores the join attributes $W_k(R_i)$ of the results of a join operation. Inner nodes do not store any payload information in the results. This reduces redundancy since payload columns are not needed for actual join computation and saves costs to storage. Also any intermediate join results that will eventually fail a later join condition are not fully computed. This reduces the impact unnecessary partial results have on the overall join process.

$$S_{cost-late}(m) = \begin{cases} |R| * \{W_k(R) + W_p(R)\} & \text{if m} \in \text{leaf nodes} \\ F(m) + S_{cost}(\text{m.left}) + S_{cost}(\text{m.right}) & \text{if m} \in \text{inner nodes} \end{cases} \tag{4.3a}$$

$$F(m) = \prod_{\substack{i=1 \\ j=1}}^{\substack{i=a-1 \\ j=a-1}} f_{i,j} * \prod_{1 \leqslant i \leqslant a} |R_i| * \sum_{i=1}^{a} W_k(R_i) \tag{4.3b}$$

The communication costs for the late materialized approach are modeled in Equation 4.4a. Our model dictates that, when possible, only the necessary join attributes are communicated within nodes in the topology. Payload information of the relations should be moved about as less as possible. In adherence to this requirement, $C_{cost-late}(m)$ defines that both inner and leaf nodes always transfer join attributes $W_k(R)$ only.

$$C_{cost-late}(m) = \begin{cases} |R| * W_k(R) & \text{if m} \in \text{leaf nodes} \\ G(m) + C_{cost}(\text{m.left}) + S_{cost}(\text{m.right}) & \text{if m} \in \text{inner nodes} \end{cases} \tag{4.4a}$$

$$G(m) = \prod_{\substack{i=1 \\ j=1}}^{\substack{i=a-1 \\ j=a-1}} f_{i,j} * \prod_{1 \leqslant i \leqslant a} |R_i| * \sum_{i=1}^{a} W_k(R_i) \tag{4.4b}$$

Reconstruction Costs for Late Materialization We have seen earlier in Section 4.5.1 that when using the late materialized approach the final result that is obtained after traversing the full join plan is simply a list of identifiers. Each element in this list maps to a tuple from one of the relations involved in a join plan. In more traditional terms we can say that this is a list of primary keys. It is obvious that this is not the desired result since none of the actual tuples are present here. So now the idea is to traverse all the relation stores, which are the leaf level nodes in our tree representation (Figure 4.7), and fetch the appropriate tuple from each store to eventually obtain the final result. Since this process involves network communication, we define a generalized reconstruction cost model $RECON_{cost}$ in Equation 4.5.

$$RECON_{cost}(\sigma) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} (W_k(R_j) + W_p(R_j)) + \sum_{m=i+1}^{n} W_k(R_m) \right) \tag{4.5}$$

Given a total of n relations involved in join plan and a reconstruction order given by σ, we start the reconstruction process at a relation R_i. Here the payload information for R_i is fetched and attached to the result. This along with identifier information for the remaining $n-1$ relations are now forwarded on to the next store. The process continues until all the relation stores have been traversed. The order in which this traversal takes place does not affect the final result. As any permutation of the n available relations can be used to carry out the reconstruction, we define the following optimization goal:

$$\min_{\sigma} RECON_{cost}(\sigma) \tag{4.6}$$

In Equation 4.6 we consider σ as the set of all possible permutations of the relations $(R_1, ..., R_n)$. And the optimization is such that we pick the specific permutation that gives us the minimum reconstruction cost on every step. Ideally this permutation would be the one which orders the relations in order of ascending payload. Even though the total cost of reconstruction still remains the same, using this optimization we can keep the size of the result set at a minimum every step because we are fetching the tuple which at that moment has the lowest payload size.

4.7 Modelling Issues with Multi Way Joins

Modeling storage costs for multi way joins conceptually still remains the same. Instead of adding the costs of the left and right nodes only, which we do in case of binary trees, we would rather sum over all the child nodes of a given node in order to obtain its storage cost. But modeling the communication cost is not as straight forward. Consider the example multiway join operator in Figure 4.8. This operator is responsible for the join operation between three relations R_1, R_2, and R_3 with the join conditions being $R_1.a = R_2.b$ and $R_2.b = R_3.d$. In this case simple bi-directional probing between the relations will not work

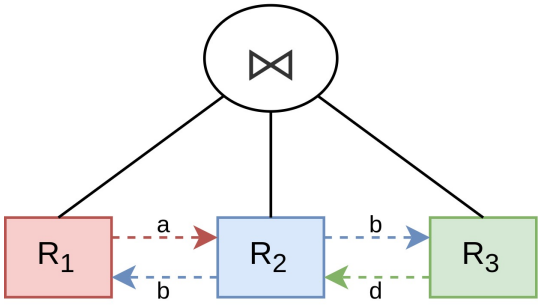


Figure 4.8: Multi-way join operator with 3 relations

anymore. This is because there are nested intermediate joins present within this this multi-way join operator itself. These are the results of $R_1 \bowtie R_2$ and $R_2 \bowtie R_3$ respectively. These nested intermediate results are never materialized. So every time a tuple from the third relation performs a probe operation, the corresponding intermediates need to be recomputed again. This will increase the amount of communication that takes place within the operator. When we have even more relations being joined under one operator, the number of nested intermediates will be even higher and the communication complexity will also increase accordingly.

In our generalized model we have worked with binary join trees in order to avoid the above mentioned complexities. But this does not mean that the late materialization approach is not applicable for multi way join operators. The concept is still going to be exactly the same. All probing operations will involve transfer of join attributes only. The payload set will not be communicated at any stage of the join computation process. Once the join result is computed, the resulting set of identifiers can simply traverse all the relation stores to obtain the full final result.

4.8 Effectiveness of Late Materialization

We have established that late materialization of join results in streaming joins can yield performance benefits by having to store and communicate less information during join computation. Intermediate join stores are a prime example of this. Having only the join-attributes of the relations saved in these intermediary stores yields the benefit of not spending too much resources on intermediate joins which eventually do not materialize to a final result.

For example, say when performing a join operation between five data streams, we obtain intermediate results for few tuples which satisfy the join predicates for four of the streams involved. But, the last predicate which involved the fifth stream fails. In this case, if early materialization is used, storage and communication resources will be spent in full in order to maintain those intermediate results. Instead if we materialize the results lazily and only store the join-attributes for the intermediate stores it becomes less of an overhead.

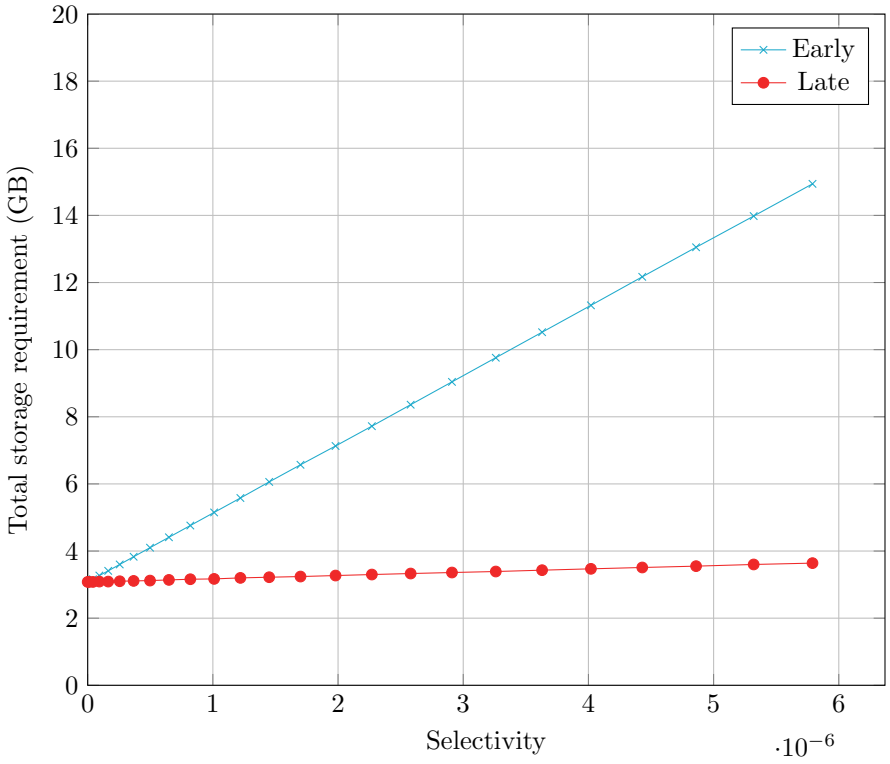


Figure 4.9: Storage cost comparison

The reconstruction of the join results though can sometimes hamper performance. High selectivity, large number of input streams or very high average payloads for all input streams will all slow down the reconstruction process. Therefore we will now discuss how storage and communication costs vary with increasing values of parameters such as selectivity, total tuples processed and payload/key size for both early and late materialized approaches. We will use our base topology and its corresponding cost models to compute costs for a range of parameter values. Since the cost models already include reconstruction cost, we will be able to determine a threshold point or range of values within which late materialization is actually beneficial. We will also demonstrate some additional cases were late materialization has very clear advantages but only when there is no trade off with the reconstruction performance, i.e. cases which will be applicable only when the reconstruction is decoupled to a separate system for example.

4.8.1 Storage Costs

Figure 4.9 illustrates the storage requirements for early and late materialized results for increasing selectivity values. The selectivity values in the x-axis are the product of all three selectivities. We keep the number of tuples observed constant at 1000 for eatch relation and the join-attribute and payload sizes

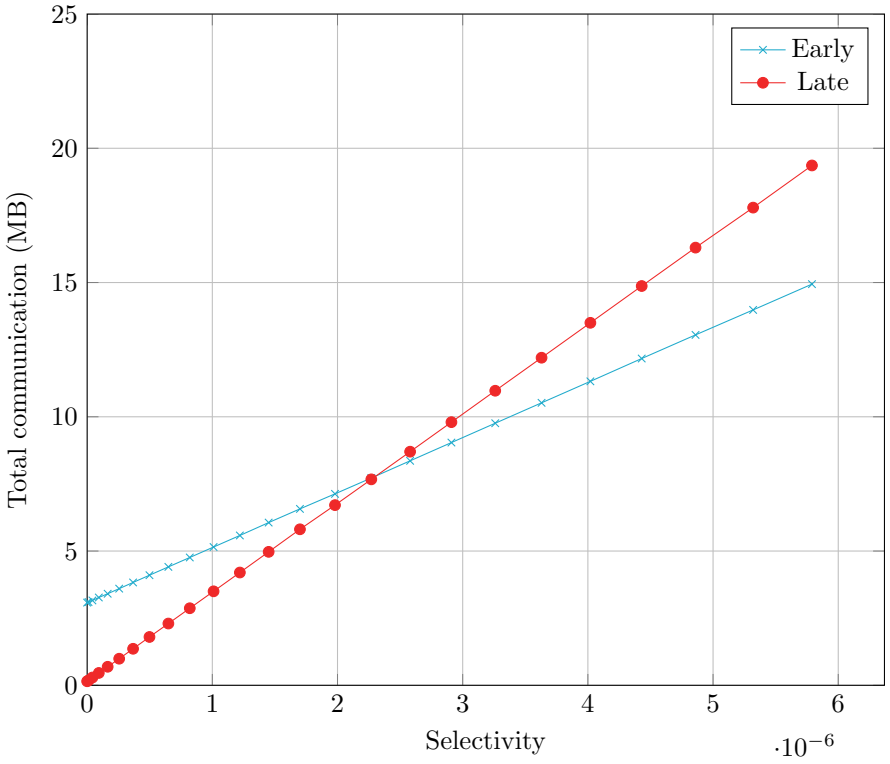


Figure 4.10: Communication cost comparison

constant at 0.05 KB and 1 KB respectively. The selectivity values of $f_{R1,R2}$ start at $1 * 10^{-6}$ and goes upto $2.40 * 10^{-3}$ and those of $f_{R1,R3}$ start at $1 * 10^{-5}$ and goes upto $2.41 * 10^{-3}$. The major difference that can be observed here between the two approaches is the growth of storage requirements. The rate at which storage requirements grow for early materialization is much higher then that of late materialization. This is due to the fact that when the results are eagerly materialized, full tuples of the intermediate join $R1 \bowtie R2$ are stored. But for late materialization we only need to store the join-attribtues for these intermediate results. This is what leads to the difference between the storage requirements with increasing selectivities. In the first section of the graph we notice that both approaches require almost the same amount of storage. But as selectivity increases, more and more intermediate results are observed leading to greater storage costs. It is worth mentioning here that even though we see a huge improvement in storage requirements, with increasing selectivities there also comes the need for a much greater amount of reconstruction. This is one of the trade off scenarios that was mentioned before. At higher selectivities we do achieve large benefits in storage costs when using late materialization, but communication costs for reconstruction must also be kept in consideration before selecting an approach.

4.8.2 Communication Costs

Figure 4.10 shows the variation in communication costs for the two approaches for increasing selectivity values. Here the selectivity values of $f_{R1,R2}$ and $f_{R1,R3}$ are the same as for Figure 4.9. The number of tuples observed for each relation is kept at 1000, join-attribute size is 0.05 KB and payload size is 1 KB.

We notice here that there is an intersection between the two lines, after which late materialization starts to deteriorate in performance. This is expected because a large number of observed join results will also mean greater tuple reconstruction costs. At lower selectivity values we see that late materialization is superior to early materialization. This is because the intermediate join results which do not eventually form final join results are never fully materialized. By avoiding the full materialization of unnecessary intermediate results we are able to save communication costs within the topology.

It should be noted that the proposed late materialization approach does not avoid unnecessary join computations but instead, minimizes their impact on the topology. This makes sense because when computing a join query over 4-5 data streams (or more), it is not possible to determine whether or not computation of further intermediate results are necessary or not until the entire join plan has not yet been traversed. Therefore by only storing the join-attributes for intermediate results we are able to reduce the impact of communicating these over the topology. This indirectly decreases the adverse effect of irrelevant intermediates.

We present an interesting scenario in Figure 4.11 where the selectivity values of both $f_{R1,R2}$ and $f_{R1,R3}$ is $1/|R1|$. For each relation we start at 1000 tuples and go all the way upto 13000 tuples. The selectivity value setup mentioned here is very common and is observed a lot in real world schema designs. Assume three relations *Person, Nationality, Address* where the *Person* relation contains foreign key columns for both *Nationality* and *Address*. Also assume that a *Person* tuple must have only one *Address* and only one *Nationality*. Under these circumstances, the query below,

```
SELECT * FROM Person p, Nationality n, Address a
WHERE p.nationality_id = n.id
AND p.address_id = a.id
```

will always return $|Person|$ many results. The above query, even though very basic, provides us with an upper bound to performance. If we use other comparisons such as non-equality ones, or even restrict the normalized tables to one value such as one single address or nationality, the number of results will not exceed $|Person|$ but will actually be less. This example scenario is what we are modeling in Figure 4.11. For streaming relations which follow such a design, late materialization is seen to be constantly beneficial. If we allow 1:n ratios for the normalized relations and also consider theta joins in generals rather than just equi joins, the upper bound is not as simple to determine. The scope of this work is within equi joins and therefore we leave that investigation to future work on improvement of the established late materialization model.

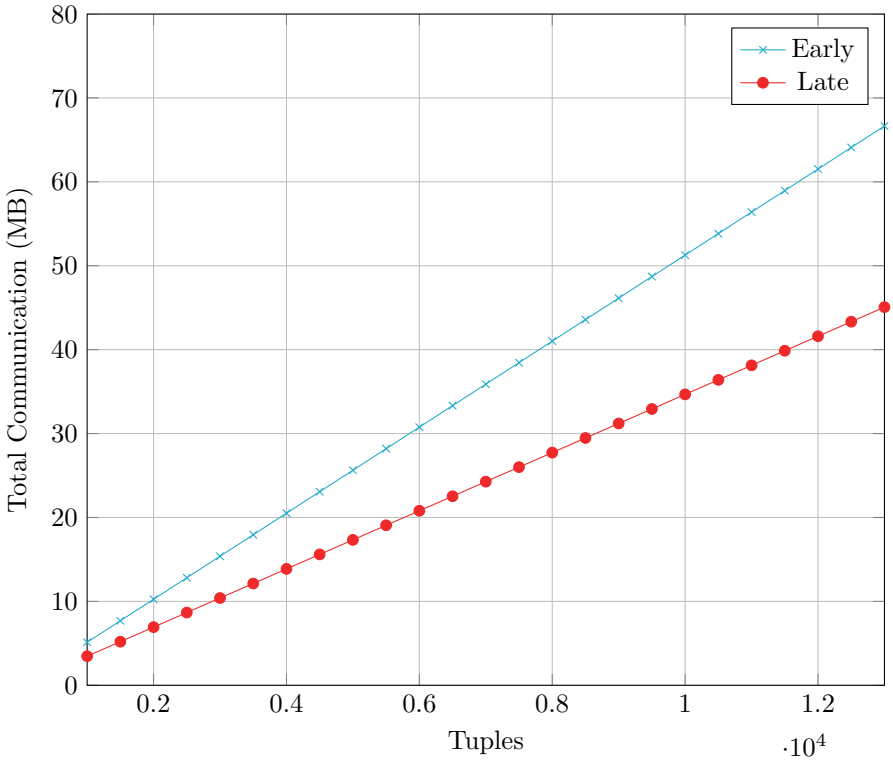


Figure 4.11: Communication costs with selectivity as $1/|R_n|$

4.8.3 Effect of Increasing Payload

The previous comparisons have been shown for varying number of tuples and predicate selectivities. We will now demonstrate the effect of increasing payload sizes on the performance. We introduced the payload concept earlier as being a set of all columns of a schema which are non-join attributes, i.e. not required for join computation. Low predicate selectivity and high payload sizes are conditions where late materialization can provide better performance. This is because its an ideal situation for lazy materialization of results. Firstly, a low predicate selectivity will mean that number of join results will be low. This allows the reconstruction phase to be much faster as fewer intermediate joins will eventually have to be reconstructed to full results. Secondly, higher payloads mean that early materialization will suffer from high communication costs during join computation. But late materialization is independent of payload size during join computation since it only works with the join-attributes which are generally very small in size (primary/foreign keys or a very few selected columns). And since selectivity is assumed to be low, the reconstruction will not be heavily effected by the large payload as well.

The graph in Figure 4.12 shows the variation in total communication cost for increasing payloads. Here we set $|R1|$, $|R2|$, and $|R3|$ at 1000 tuples each. $f_{R1,R2}$ is 0.0001 and $f_{R1,R3}$ is 0.01. We vary the payload sizes from 1 KB all the way to

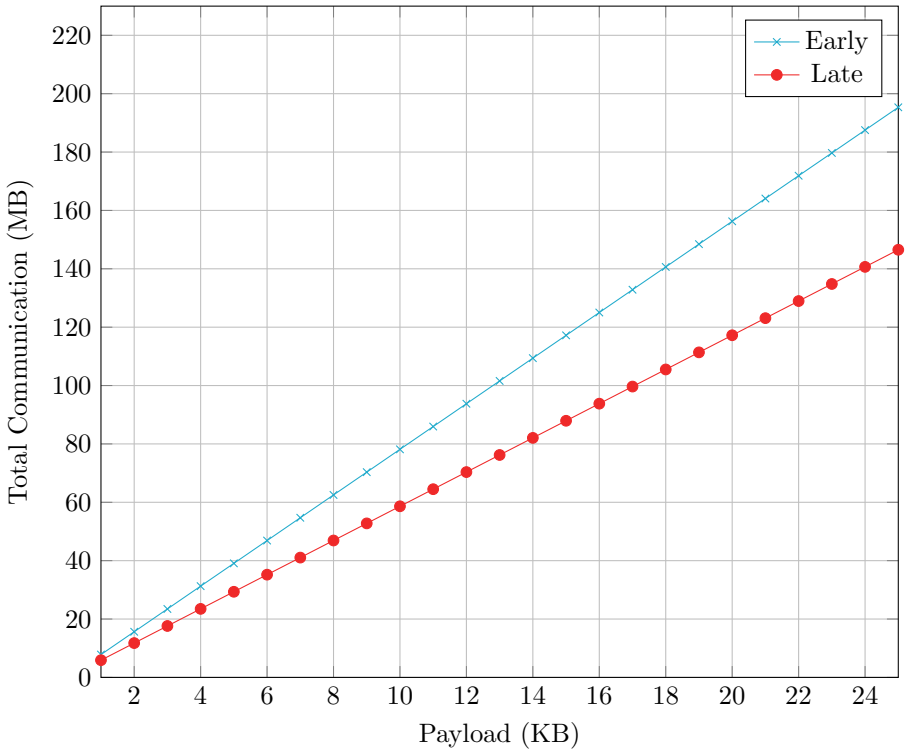


Figure 4.12: Communication cost comparison for increasing payload

25 KB. The join-attribute size is kept constant at 0.05 KB. Initially the difference between the two approaches is negligible. But as we keep increasing the payload it is seen that late materialization starts to perform much better. It computes all the expected results with much less amount of communication within the topology. It is important to explain the sensitivity of the parameters in this case. For example, keeping the selectivity values as is and increasing the number of observed tuples to 50000 or 100000 will alter this result completely. Then we will see that even for an increasing payload, early materialization is performing better than late. This is again due to the reconstruction overhead. Number of observed tuples has increased but the selectivity remains the same meaning there will be a much larger number of results. This coupled with high payloads means that the reconstruction phase will incur very high communication costs before emitting the final results. In such a scenario it becomes an overhead.

4.8.4 General Applicability of Late Materialization

When dealing with streaming data it is not possible to have traditional query plans and statistics as with relational data. Since the data is not stored or indexed, standard statistics for individual relations are not available [13]. This makes it difficult to estimate performance based on the models presented here. But in general, based on the numbers observed from the model, we can assume the following:

1. Late materialization is most affected by total number of join results. If the selectivity values are high, then a large part of the input tuples will eventually form join results. In this case it is possible that cost savings in terms of communication might be exceeded by the reconstruction cost. So for a low number of input tuples, selectivity does not have much effect. Rather payload size has a greater impact here. But for large input sizes, low selectivity is crucial for late materialization to perform better.

2. If the streams are generating input tuples at a high rate, the topology will have an advantage by using late materialization since it will only use the join-attributes for computing join results. The fast computation of results will allow the topology to be free to accept more incoming tuples. But it should be noted that in this case decoupling of the reconstruction process might be required because it may cause a bottleneck.

3. In cases where the reconstruction phase can be decoupled into a separate topology, the range of applicability of late materialization will be much higher. Join predicates with much higher selectivities can be used. Payload also in this case will not be an issue since its only going to be part of the reconstruction phase and is going to be handled separately from the join computation phase.

Any stream processing system implementing late materialization of join results should be able to switch between late and early materialization dynamically for optimum performance. This is mainly due to the unpredictable nature of streaming data. For example, by computing selectivities over the observed tuples and results and monitoring average payload values for the involved relations till current time point T, the topology can determine whether or not late materialization is being beneficial using the cost models defined here and hence switch between eager and lazy materialization as needed.

Chapter 5

Implementation

We have implemented the base topology presented in Chapter 4 using Apache Storm [4] along with late and early result materialization capabilities. In this section we will discuss the data sources and additional tools used to implement the experimental models. The structures of the topologies for different types of materialization have been elaborated. We also present some extensions to the base topology which were required to obtain better results in favor of late materialization.

5.1 Tools and Data Sources

In order to test the approaches introduced in the previous section, we have implemented the base topology along with late and early materialization capabilities. Since our approach to join computation is inspired by the Multistream Join operator presented by Dossinger et al. Michel [16], it made sense to utilize the same tools and programming languages that have been used to implement CLASH. Hence we have used Apache Storm [4], which is a distributed stream processing framework capable of real time processing of continuous data streams, as a base framework with Kotlin as the programing language of choice.

We use synthetic data generated based on the TPC-H specification as our data source. In Table 5.1, details of the relations used for the experiments are given. For simplicity the reference names mentioned for each relation will be used instead of the full names in this document. At the moment only the 3-way join defined by the base topology has been implemented. Note that there is no restriction to the number of relations that maybe joined. More relations can be added by simply implementing the additional bolts and stores within the topology. Topologies are generated dynamically in CLASH and as it is not within the scope of this work we will only implement the late materialization model and compare its performance with early materialization in manually constructed topologies.

Each of the spouts are initially populated with a result set from the TPC-H schema. We have used the line item, order and supplier tables for our experiments. An example of the queries used to build the result sets and initiate the spouts for each of the relations are shown below. In order to obtain join results faster it was necessary to sort both line item and order tuples by the **order key** in the same ascending order. It should be noted here that not ordering the results in these select queries would actually be a more accurate simulation of a streaming environment. This is because join partners would arrive randomly and certain tuples may have to wait a long time for their join partners to arrive. Also since we are restricting the number of tuples and not selecting the whole table, if no ordering is applied the number of join results could be very low

Relation	Reference name	Number of Tuples	Number of Columns
Line Items	**li**	6001215	16
Orders	**o**	1500000	9
Suppliers	**s**	10000	7

Table 5.1: Table of relations used as streaming sources

because most join partners may be left out during the selection filtering. This is not ideal because a low number of join results would not give a good indication of how the two models are performing differently.

Line Items:

```
SELECT * FROM lineitem ORDER BY l_orderkey ASC LIMIT 200000
```

Orders:

```
SELECT * FROM orders ORDER BY o_orderkey ASC LIMIT 200000
```

Suppliers:

```
SELECT * FROM supplier ORDER BY s_suppkey ASC
```

5.2 Statistics Collection

We can use various time based metrics to determine the performance of each materialization method. In the implementation we use an additional bolt called the Ticker bolt. The two tasks of this bolt are, to log the time when an input tuple first enters the topology and to log the time when a result is emitted out of the topology. Using these log times we are able to determine various statistics such as overall duration etc. In addition to logging the timestamp, the ticker bolt also logs the source of the tuple and also its type. The source of the tuple is simply the ID of the bolt or spout it was emitted from. The tuple type is used to distinguish between input tuples and final join results. In addition to times the log data from the ticker bolt also allows us to check if the join operation was completed successfully by checking if the number of observed final results is the same as expected. Each log entry of the Ticker bolt consists of the following attributes:

- **id:** Unique identifier.
- **source:** The node from which this emit occurred.
- **type:** The type of emit, which is the flag accompanying every tuple (Table 2.2)
- **log_time:** The time at which the tuple was emitted.

Figure 5.1 illustrates the base topology including a ticker bolt. Note that the two ticker bolts shown in the diagram are only for ease of understanding. In the actual implementation there is only one such bolt. The final results before being emitted out of the system are tagged with the timestamp at which they reached

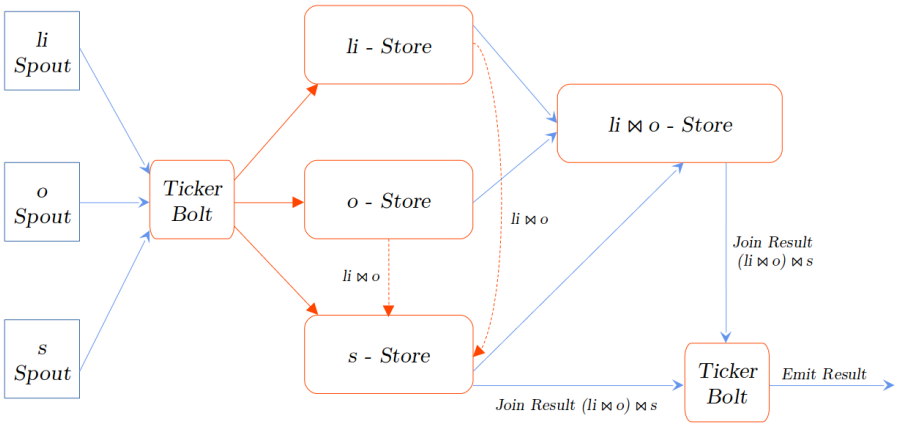


Figure 5.1: Base topology with early materialization and ticker bolt

the ticker bolt. This tells us that at what exact time the result was created. Another difference that needs to be pointed out is that the ticker bolt now also handles the communication necessary for probing other stores for potential join partners. For example, When a *li-tuple* arrives at the ticker bolt it will forward this to the *li-store* as well as the *o-store*. This way both storage and probing are covered.

5.3 Early Materialization

Early materialization in the base topology is implemented just as it was explained in the approach. *li* and *o* tuples probe each others stores upon arrival and generate $li \bowtie o$ intermediate results which are stored in its respective store. Depending on what arrives first a result tuple can be observed from either *s-store* or the $li \bowtie o$ store. Finally the results go through the ticker bolt where these are logged and emitted. The $li \bowtie o$ store contains the full *li* and *o* tuples (keys+payload). Because of this it is possible to emit the result directly and no additional reconstruction is required.

5.4 Late Materialization

Both late and early materialization approaches can make use of the same base topology or any topology for that matter. The primary implementation difference arises in what information is communicated within the topology during join computation and in what order the reconstruction is of results is performed.

For late materialization all node to node communication within the topology during join computation involves only the join-attributes. The payload is not included in any of these. The need to transfer payloads along with results only comes up during reconstruction of the results.

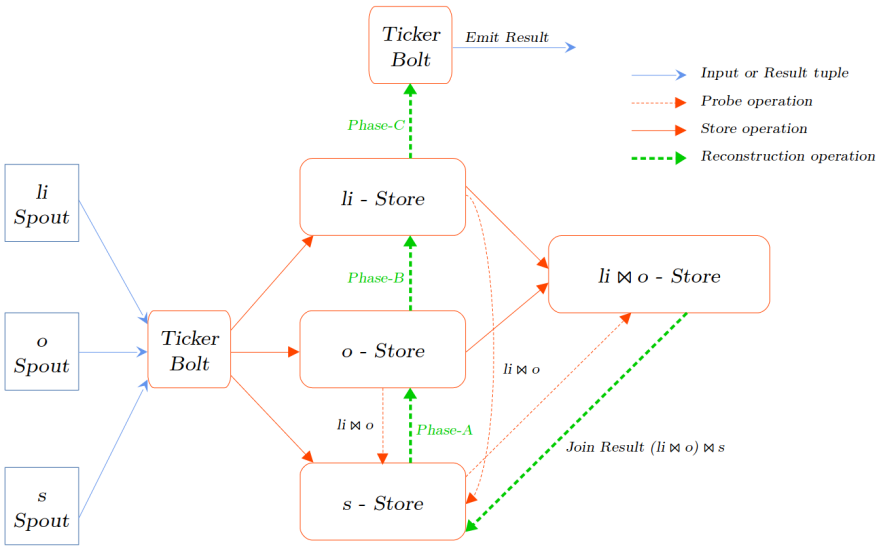


Figure 5.2: Base topology with late materialization and ticker bolt

In our implementation, the reconstruction order is $s \rightarrow o \rightarrow li$. Ideally the reconstruction order would be determined dynamically based on ascending order of average payload of the relations. Since for our experiments we are dealing with a very few number of relations in the join operation, this order can be hard coded. The ordering mentioned above is following this approach. The supplier tuple's payload is the smallest, followed by the order, followed by the line item which has the heaviest payload. This is even more so because we will later add additional payload data to the line item to make it even larger. With the predefined reconstruction order, it is important that all final join results (only sets of identifiers, not full tuples) end up in the $s - store$. This is because the reconstruction process is initiated there. Since it is possible for join results to be generated in the $li \bowtie o$ store, these must first be transferred to the $s - store$ and then sent out for reconstruction. We must keep this implementation issue in mind as well. It should also be mentioned that decoupling the reconstruction phase will eliminate this issue. This is because the reconstruction is done in a separate topology, it can use the appropriate reconstruction order and go through each of the stores as needed.

Figure 5.2 shows the implementation of late materialization in the base topology along with the ticker bolt. The procedure for generating intermediate $li \bowtie o$ results is the same as for the early materialized implementation. Only difference is that here the tuples in $li \bowtie o$ store only contain $li - key, o - key, s - key$ and no payloads. From here we elaborate how identifier only join results are obtained and how these are further reconstructed.

1. If a s tuple successfully probes the $li \bowtie o$ store, the join result is forwarded back to the $s - store$ and reconstruction is initiated.

2. If a join result is found in the $s - store$ itself, reconstruction can be started

directly

3. In *Phase - A* a full s tuple of a join result along with identifiers/keys of corresponding o and li tuples are transferred to the $o - store$.
4. In the $o - store$, the corresponding o tuple from the *Phase - A* is attached to the result. *Phase - B* is now the transfer of a full s and o tuple along with the key of li tuple to the $li - store$.
5. In the $li - store$ the corresponding li tuple from *Phase - B* is attached to the result and emitted directly.
6. Finally *Phase - C* is the emit of the fully constructed result tuple

In this manner all result tuples are generated and reconstructed. An important note here is that *Phase - C* is not included as part of the communication cost model for late materialization. This is because here the result is obtained and emitted directly and it is not a node to node communication.

From the topology in Figure 5.2 we can notice that there is a high amount of node to node communication when we include the reconstruction phases. Storm is designed to have output fields for a bolt/spout predefined before the topology is deployed. And these field definitions are generally fixed. It is possible to have *named streams* were each stream has its own ID and fields can be defined for each unique stream name. This feature is used to define fields in the ticker bolt such that it routes the tuples properly to cover both storage and probing. But this named streams approach is complicated to implement, especially when the topology is very large. Distinguishing which part of the join computation process a tuple belongs to is crucial for correct operation. For example, tuples which are coming in for reconstruction should be treated accordingly and not used for join computation. While the opposite is true for new incoming tuples or intermediate joins. In order to be able to distinguish and separately process the tuples being exchanged between nodes a *Flag Field* was also emitted along with the other fields of the tuples. Table 5.2 outlines the details of the flags and how these served the given purpose. The flag is used to identify exactly which part of the process a particular tuple is in and then handle it accordingly.

Experimenting with just the base toplogy and regular tuples streaming from TPC-H sources did not yield convincing results for the late materialized approach. This is because effectiveness of late materialization depends on whether or not we receive greater benefits in terms of storage and communication costs compared to the reconstruction overhead. The results from the initial set of experiments confirmed that in order to see some benefits from late materialization it is necessary to make changes which will put more load on the communication and storage requirements of the topology.

5.5 Larger Payload

Using larger payloads, we encountered a couple of issues when running the experiments that was not thought of before hand. When 3 kilobytes of additional payload is added to the *lineitem* tuples this very quickly exceeds the memory

Flag	Description
INPUT_TUPLE	Used to identify tuples which have just been emitted from spouts and still have not found join partners
INTERMEDIATE_JOIN	Partial join result. This is true for all partial join results existing until the whole join tree is fully traversed.
FINAL_JOIN	A complete join result. This contains all the tuples of the relations present in the join.
MATERIALIZATION_RESULT	This is also a complete join result but contains only the join keys. Whenever this is encountered reconstruction is performed.

Table 5.2: Flags used to distinguish tuples being exchanged within nodes

capacity of the node. Assume that we expect 2 million *lineitem* tuples as input each having a payload of 3 kilobytes. Once all the *lineitem* tuples have been emitted by the corresponding spout, the size of the store would be approximately 6 gigabytes. It is not possible for one instance of the given store to store and process all of this tuples. Even if we increase the Java heap size for that worker node that is still not a practical solution. The ideal solution for this would be to have multiple instances of the store running in parallel. So for all experiments involving additional payloads, the bolts in our topologies had 6 tasks each. This way the 2 million *lineitem* tuples would be divided almost equally among the tasks via Storm's **fieldsGrouping** capability. Fields grouping sends input tuples to tasks of a bolt based on the value of one or more selected fields. For example, Figure 5.3 illustrates the extended base topology where *lineitem* tuples are joined with *supplier* tuples in one half of the total join process. In this case we would send *lineitem* tuples to both *lineitem* and *supplier* stores based on the supplier-key. The *supplier* tuples would also use the supplier-key for grouping as well. This would ensure that tuples with the same supplier key end up in the same task and produce join results.

The second issue was missing a large chunk of final join results when using late materialization with additional payload. The number of results that were being missed seemed to grow as more tuples were being processed. This was due to the fact that some reconstruction requests of full join results were arriving at the *lineitem* store well before the *lineitem* tuples that are part of those results were successfully saved in the store. The early result generation was possible because the *TickerBolt* forwards the *lineitem* tuples to two stores at the same time (*lineitem* store for storing and *order* store for probing with regards to the base topology). The probe operation of the *lineitem* tuple was creating results very quickly and these were failing eventual tuple reconstruction in the *lineitem* store. We solved this problem by changing the probe mechanism. When using late materialization *lineitem* spout does not probe the order store. This is now switched to the *lineitem* store. This guarantees that a *lineitem* tuple will only probe the order store once it has been saved therefore guaranteeing successful reconstruction.

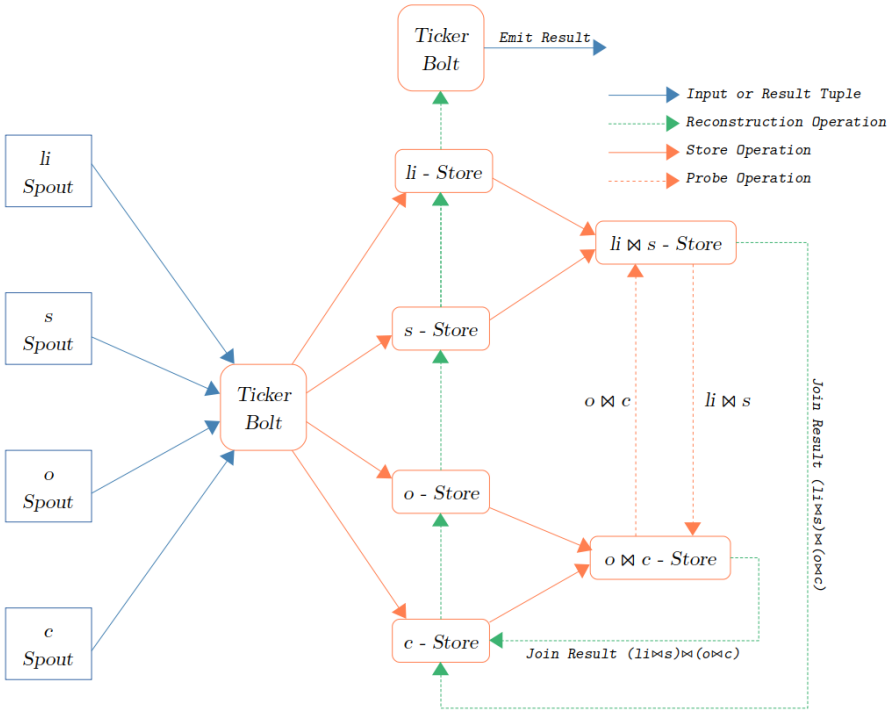


Figure 5.3: Extended base topology with four relations

5.6 Extended base topology

The second approach involves adding another relation to our join plan. The original base topology computed a join query involving the relations *lineitems*, *orders* and *suppliers*. We now add the *customers*(c) relation to the join query as well. This causes some changes in the base topology. Using the four relations it is now possible to have two materialized intermediate join results. The $li \bowtie s$ and $o \bowtie c$ intermediate results will be materialized and have their own stores. The idea behind doing this is to increase the amount of inter-topology communication to a level where benefits of late materialization can be clearly observed. This extended base topology (Figure 5.3) will now have a different tuple reconstruction work flow. Results can be emitted from either the $li \bowtie s$ store or the $o \bowtie c$ store and are forwarded to the *customer* store to initiate result reconstruction. The reconstruction flow is as follows: $c \rightarrow o \rightarrow s \rightarrow li$. The reconstruction procedure is still the same, we start from one store, fetch the corresponding tuple and continue on to the next store until all stores have been traversed.

Chapter 6

Experiments

6.1 Setup

For the purpose of establishing the hypothesis it is required to prove that late materialization can yield better performance in some cases when compared to early materialization. It is assumed to be most effective when tuples from incoming streams are very large in size or the rate at which tuples are arriving is very high. Therefore, a comparison has been made between the performance statistics of the early materialized model vs. the late materialized model. In this case, it is not important to compare it with another stream processing framework or something similar because the goal is to prove that there is a performance difference between the two approaches. Since these approaches can be implemented in any framework, it's effect does not need to be considered.

The topologies described in the previous sections have been implemented in **Apache Storm** using **Kotlin** as a programming language. Since relational data has been used to emulate a data stream, the total number of tuples emitted by a relation was controlled and steadily increased in order to observe how processing times vary along with it. Since a large number of line item and order tuples are present (Table 2.1), initially the experiment was started with 500000 tuples each of both line items and orders and increased all the way upto 6 million tuples for line items and 1.5 million tuples for orders. The number of suppliers and customers were always constant at 10000 and 150000 respectively. In order to determine performance various duration based statistics were gathered while the topology ran.

- **Effective Duration:** This is the time between the first observed join result and last observed join result.
- **Actual Duration:** This is the time between the first and last tuple processed within the topology.
- **Result Delay:** This is the time between the first tuple processed and the first result that is observed.

Statistics Collection To determine the durations mentioned above we tagged tuples entering and leaving the topology with **timestamps**. A separate bolt named **Ticker Bolt** was used to do this. The primary function of this bolt was to store log entries for all the tuples being processed within the topology (input tuples and full results). All incoming tuples from spouts first go through the ticker bolt where these are tagged with the current timestamp and then logged into a database. The same is true for final results. All final result tuples go through the ticker bolt where these are tagged and logged. These log entries

were later queried to compute the required statistics. Each log entry consisted of the following attributes:

- **id:** Unique identifier.
- **source:** The node from which this emit occurred.
- **type:** The type of emit, which is the flag accompanying every tuple (Table 2.2)
- **log_time:** The time at which the tuple was emitted.

The following queries were used to calculate the processing times. **PostgreSQL** was used to store the log entries.

Effective Duration:

```
SELECT (MAX(log_time) - MIN(log_time)) AS firstResultToLastResult
FROM ticker
WHERE type = 'FINAL_JOIN';
```

Actual Duration:

```
SELECT (
        (SELECT MAX(log_time) FROM ticker) - (SELECT MIN(log_time) FROM
            ticker)
) AS totalDuration;
```

Result Delay:

```
SELECT (
        (SELECT MIN(log_time) FROM ticker WHERE type = 'FINAL_JOIN') -
            (SELECT MIN(log_time) FROM ticker)
) AS firstTupleFirstResultDuration;
```

Spout Initialization and Expected Results For our experiments the spouts were initialized by first querying the TPC-H database and obtaining result sets for the desired relations and then emitting each tuple into the appropriate stream. Experiments were performed for both the base topology and also the extended base topology. We came up with equivalent SQL join queries for our topology operations and ran them to determine the expected result counts. Examples of such a query for 1000 line items and orders are as follows:

Expected result count for Base Topology:

```
SELECT COUNT(*) FROM
(SELECT * from supplier ORDER BY s_suppkey) s,
(SELECT * from orders ORDER BY o_orderkey LIMIT 1000) o,
(SELECT * from lineitem ORDER BY l_orderkey, l_suppkey LIMIT 1000) l
WHERE s.s_suppkey = l.l_suppkey
AND l.l_orderkey = o.o_orderkey
```

Expected result count for the Extended Base Topology:

```
SELECT COUNT(*) FROM
(SELECT * FROM customer ORDER BY c_custkey) c,
(SELECT * FROM supplier ORDER BY s_suppkey) s,
(SELECT * FROM orders ORDER BY o_orderkey LIMIT 1000) o,
(SELECT * FROM lineitem ORDER BY l_orderkey, l_suppkey LIMIT 1000) l
```

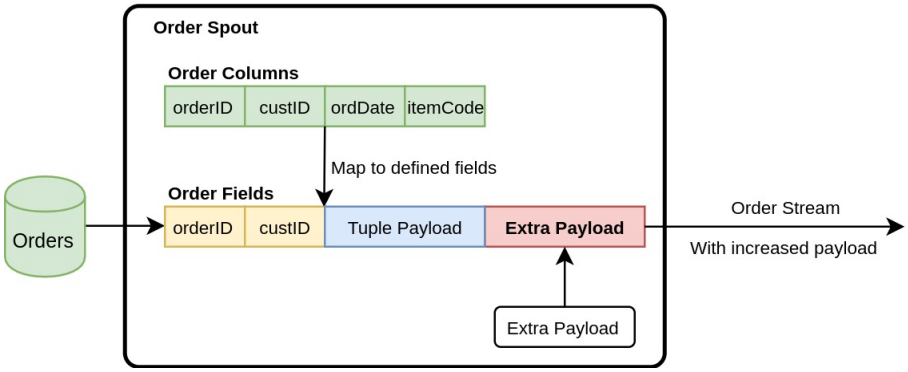


Figure 6.1: Increasing payload size of an incoming tuple

```
WHERE s.s_suppkey = l.l_suppkey
AND l.l_orderkey = o.o_orderkey
AND o.o_custkey = c.c_custkey
```

The experiments were performed on a Storm cluster deployed on a server. The cluster contains 8 dedicated worker nodes. The number of tasks or worker processes was handled directly by the cluster during deployment of the topology. No numbers were specified for this case.

We performed several sets of experiments for both and late and early materialization by changing parameters such as number of tuples processed, additional payload and degree of parallelism of bolts within the topology. Any experiments that involved using added payload also had a high degree of parallelism within the bolts because of the reasons discussed in Section 5.5.1. Some very interesting differences were observed between the results of the base and extended base topologies.The experiments are classified as follows:

- Base topology with no additional payload and six tasks per bolt
- Base topology with 3 kilobytes of additional payload and 6 tasks per bolt
- Extended Base topology with no additional payload, one and six tasks per bolt
- Extended Base topology with 3 kilobytes of additional payload and six tasks per bolt

Support for larger payloads One of the ways by which we added more load to the system is by emitting and additional **payload** field with one (or more) of the relation streams. This could have been done by adding an additional payload column to the TPC-H tables being used. When a spout reads the result set from such a modified table, emitted tuples will have a size approximately equal to the payload. This is because the other fields have a very small footprint in comparison to the payload column. We did not follow this approach, but instead increased the payload of the selected tuples from the topology itself. We went with this approach because it makes alternating experiments between payload and non-payload tuples much easier. Also since we experiment with different

payload sizes it is simpler to do it this way. When the payload option is enabled for certain relations, the spouts will create output fields with an additional payload field. When emitting a tuple this payload field will be populated with a chosen payload string. This approach is illustrated in Figure 6.1.

6.2 Results

Every experiment mentioned above was ran five times for each unique number of tuples processed to record the average run time. Steady increase in duration was observed for an increasing number of tuples. This is expected because processing more tuples would naturally take more time. We also noted a decrease in effective duration when using parallelism in our topology. We will mainly compare the effective duration for late and early materialized approaches for every run of the experiment. We have used the effective duration metric in our result presentations. Effective duration gives the time between the first and last observed result. This metric is independent of the actual duration of the topology, which is the total time between the first tuple entering and last tuple leaving the topology. The actual duration may be longer or shorter depending on how fast the spouts finish emitting input tuples or processing delays within bolts. But effective duration is not affected by this. The experiments using the base topology are *three-way* joins between the relations *lineitem*, *orders*, and *suppliers* while the *four-way* join experiments include the additional *customers* relation and use the extended base topology.

6.2.1 Three-Way Join

In the three-way join experiments, we have one intermediate join result store. The $li \triangleright\!\!\bowtie o$ result is materialized in this case (Figure 5.2). We could have also materialized $li \triangleright\!\!\bowtie s$ result in this case but chose not to do so. In our data set, number of supplier tuples is less compared to line item or order tuples (Table 5.1). This would allow the supplier spout to finish emitting input tuples much faster than the other two. Since the supplier store would be ready much earlier, all $li \triangleright\!\!\bowtie s$ results will be observed in the supplier store. Instead by materializing the $li \triangleright\!\!\bowtie o$ we ensure that intermediate results are observed in both line item and order stores which will result in greater communication for probe operations to the supplier store. This is important for our investigation because communication costs play an important role in performance of late materialization. It should be noted that no matter which of the two intermediate results we materialize, the number of intermediate results will always be $|li|$ because the line item relation contains foreign keys for both supplier and order relations. We have investigated three-way join performance with and without parallelism enabled in the topology.

Single task performance Here we check the performance of late and early materialized approaches in our three-way join experiment using *one task* per bolt. There is no parallelism applied in this case. The results are displayed in

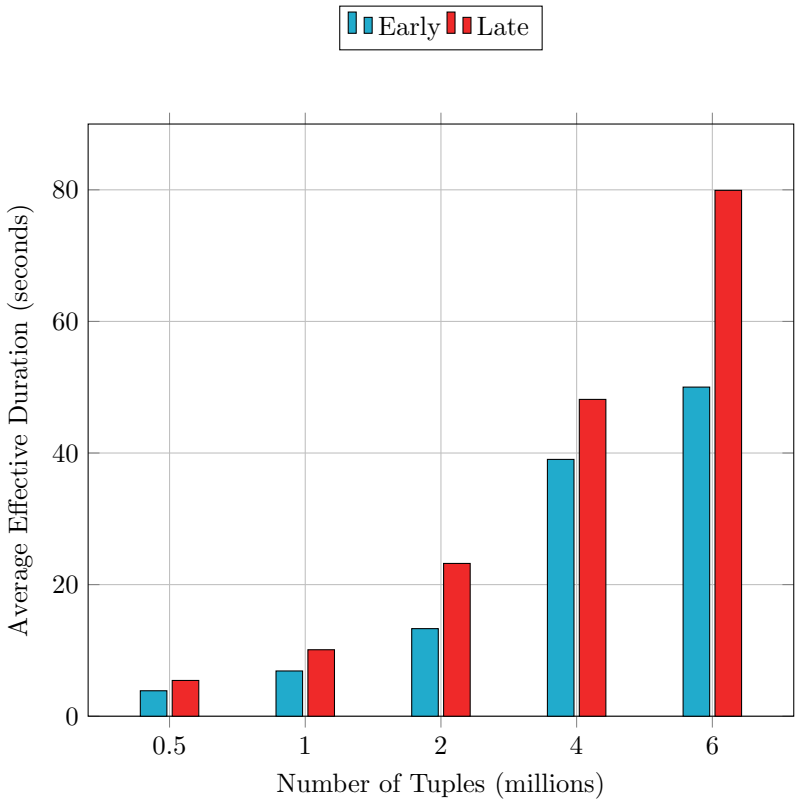
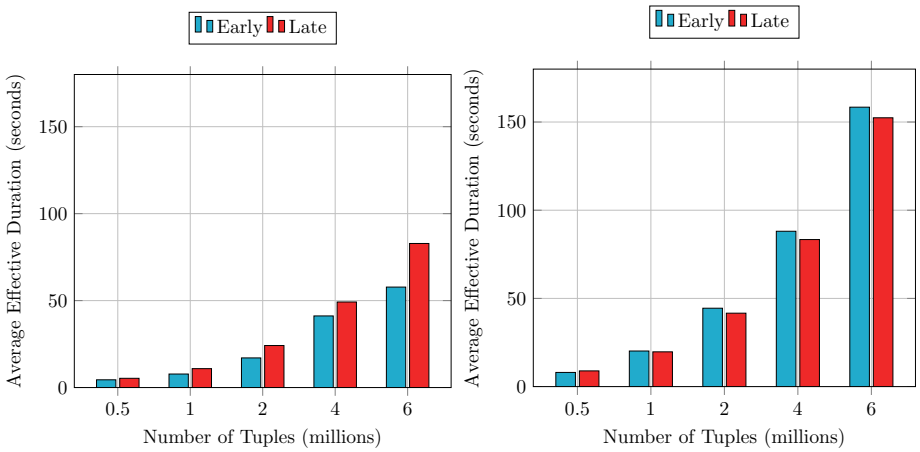


Figure 6.2: Avg. run times in base topology, tasks/bolt: 1

Figure 6.2. We notice here that early materialization is consistently better than late materialization over all input count variations. The difference between the late and early materialized approaches also increases as a greater number of input tuples are observed. This result is due to overhead of tuple reconstruction. The cost savings that we achieve during communication by using lazy materialization is much less compared to the costs incurred for tuple reconstruction. This is similar to what we see after the intersection point in our modeling evaluation in Figure 4.10 where communication costs for late materialization become more expensive than early and also grow at a faster rate.

Multitask performance In contrast to the single task experiment, here we have used *six tasks* per bolt in our topology. This level of parallelism allows us to experiment with larger payload sizes in our input tuples. Figure 6.3a shows the comparison between effective duration of late and early materialization with no added payload. Here the results are similar to our single task experiment with early materialization showing better run times. Even though the run time difference between the two approaches is still growing, the rate of growth is not as prevalent as before. We also notice that the higher degree of parallelism has not affected the overall run time significantly.

With the introduction of additional payloads, we notice the difference in per-



(a) No additional payload

(b) Additional 3KB Payload

Figure 6.3: Avg. run times in base topology, tasks/bolt: 6

formance in Figure 6.3b. Here we see that late materialization gives better run times compared to early materialization. Previously our choice of having $li \bowtie o$ materialized in order to have a greater amount of communication within the topology was not effective. Here, after increasing tuple payloads, the savings in communication costs have made a difference. Another interesting note here is that the difference in run times between late and early materialization remains approximately constant throughout the runs. This is proof of a consistent reconstruction cost incurred in every run. In this case, had the reconstruction been decoupled or performed asynchronously from the join computation process, we could have observed a much larger difference in performance.

6.2.2 Four-Way Join

The four-way join experiments are different in two ways from the three-way join ones. Firstly we have introduced another relation into the join plan which is the *customers* relation. And secondly we have two intermediate result stores instead of one. These are the $li \bowtie s$ and $o \bowtie c$ stores. The topology configuration now also changes and we have used the extended base topology Figure 5.3 for these experiments. The expectation here is that the presence of an additional relation and an intermediate result store will increase communication within the topology thus giving scope for late materialization to perform better. The choice of intermediate join results is obvious because line item tuples can be joined with either supplier or order tuples but customer tuples can be joined with order tuples only. As seen before, here we again test the two materialization approaches with and without parallelism. With parallelism enabled we also test the effect of having larger payload sizes on the performance.

Single task performance Here we check the performance of late and early materialized approaches in our four-way join experiment using *one task* per

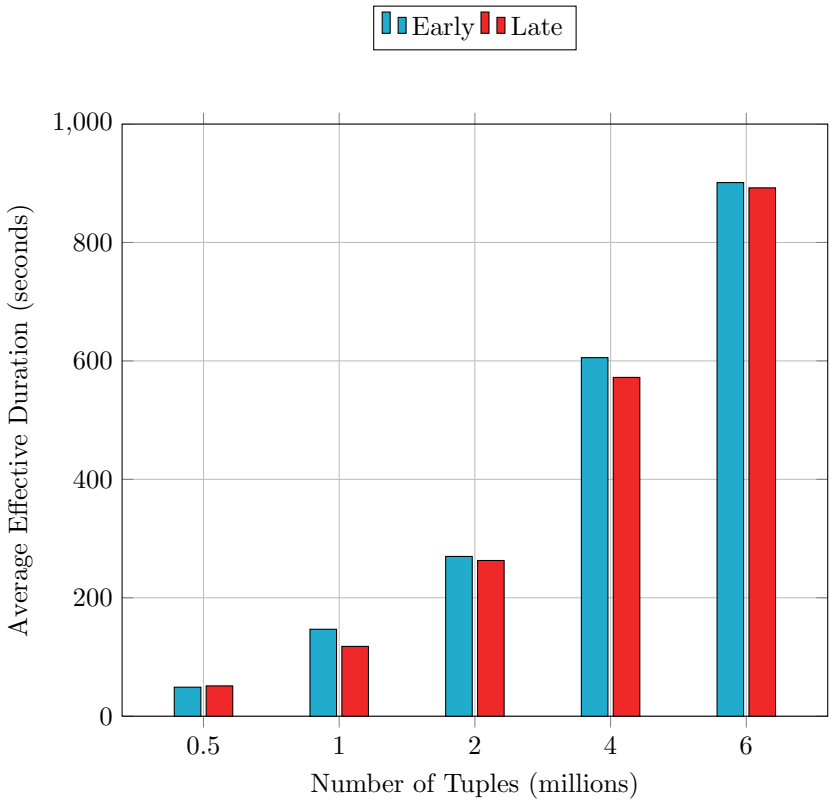
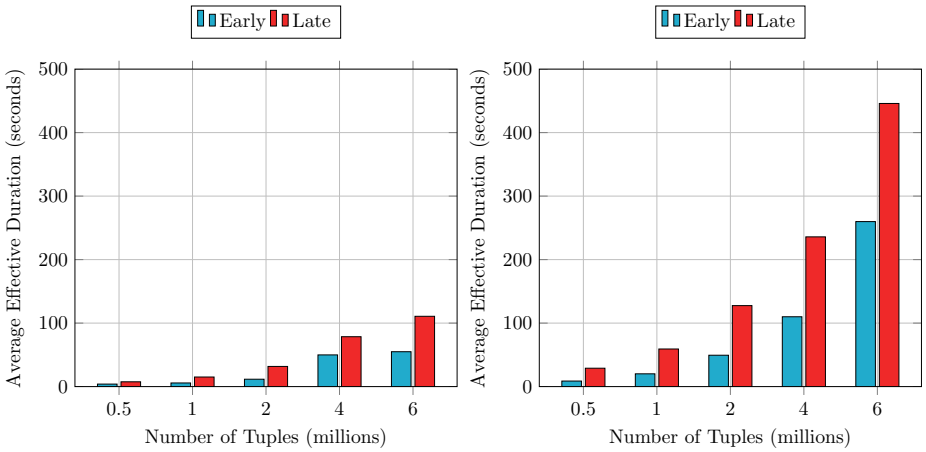


Figure 6.4: Avg. run times in extended base topology, tasks/bolt: 1

bolt. There is no parallelism applied in this case. The results are displayed in Figure 6.4. Contrary to what we have seen in the single task performance for the three-way join (Figure 6.2), here late materialization gives better run time performance compared to early materialization. This shows that late materialization will benefit from having more relations and/or intermediate stores within the join plan. This is because overall reduction in communication costs is enough to compensate the tuple reconstruction overhead.

An interesting observation here is that for the early materialized approach amount of deviation from the mean run time is much greater for higher number of tuples while that for the late materialized approach is comparatively much less (Figure 6.6). This stems from the constant reconstruction cost that is incurred with lazy materialization. Even if the time taken to get the actual join results is unevenly distributed, the reconstruction time will balance it out giving a much more uniform distribution of runs.

Multitask performance With no parallelism, using late materialization has shown to be beneficial with a join plan having several relations, atleast more than three. But this case is not the same when we increase the degree of parallelism. When the bolts are scaled out by running multiple instances, the incoming load is also divided among these, allowing the bolts to process and generate join



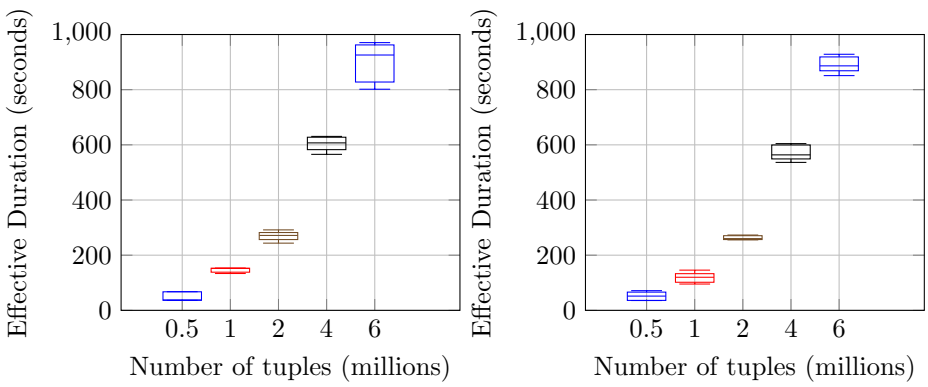
(a) No additional payload

(b) Additional 3KB payload

Figure 6.5: Avg. run times in extended base topology, tasks/bolt: 6

results much faster. We experimented with this idea where the extended Base Topology was deployed with six tasks per bolt. The results of these runs are shown in Figure 6.5. One major difference we notice here is that the overall effective duration has reduced almost by half. In this topology configuration having a high degree of parallelism is actually beneficial, this is something we did not see in the three-way join experiments. The results show that regardless of additional payload, early materialization constantly outperforms late materialization. In the four-way join experiment, results can be generated from both the intermediate stores. Having 6 tasks running of each means that there are total 12 instances from which join results can be emitted. Applying late materialization here is a good example of showing the effects of reconstruction overhead. With the high degree of parallelism results are obtained much faster compared to single tasks. So whatever communication costs we may have saved by using late materialization during the join computation is minute compared to the large overhead cost of tuple reconstruction. This allows us to reach an interesting conclusion.

Late materialized approach is not suitable when the Storm Cluster is capable of running bolts with a high degree of parallelism. But in cases of single tasks it can be beneficial. This could be possible if a Storm Cluster is deployed over a group of regular desktop grade machines each of which is not powerful enough to run several bolt tasks in parallel. On the other hand if the cluster is deployed on a server grade environment, like in our experiments, high degree of parallelism can be achieved hence late materialization actually shows adverse effects.



(a) Early

(b) Late

Figure 6.6: Run time distribution extended base topology, tasks/bolt: 1

Chapter 7

Conclusion and Future Work

In this thesis we have designed and implemented a late materialization model for stream join computation frameworks. We propose methods by which a stream processing framework can implement lazy materialization of join results by only transferring join attributes within the topology during join computation and then, at the end of the join plan, reconstruct the result tuples by fetching corresponding payload data. We also propose some optimization techniques for the reconstruction process such as appropriate reconstruction ordering and decoupling.

Using a base mathematical model we have interpreted the behavior of the late materialized approach vs. a regular early materialized approach with regards to changing selectivity values, tuple counts and payload. From this model we have found a selectivity range in which late materialization is beneficial. Stream processing frameworks can also utilize this model and compute costs and decide on whether or not to use late materialization based on the characteristics of the data it is going to process. A generalized cost model for computing storage and communication costs has also been presented. The model can be used to compute operating costs when using early/late materialization approach given an optimized join plan.

We have evaluated our model using two topologies, one of which computes join results between three relations (base topology) and the other between four (extended base topology). In our experiments we have seen that the late materialized model is effective when tuples include large payloads or there is a high number of relations present in the join plan. Late materialization shows poor performance when bolts within the Storm topology have a high degree of parallelism. This makes late materialization of join results a suitable approach when clusters are deployed in compute environments which are not capable of handling the high degree of parallelism.

There is potential for more work to be done on the presented models in order to improve and generalize them even more. Exploring a broader spectrum of data sources and implementing the models in actual stream join processing systems would be of primary importance in this case. The following additional investigations can yield better insights on the suitability of late result materialization with regards to distributed stream join processing:

- **Exploring non-foreign key joins:** In our experiments we have used the TPC-H benchmark data set which is highly suited to testing foreign key joins. Experimenting with non-foreign key joins can present new issues within our model and corresponding implementation.
- **Proper implementation:** The implementations we have presented are independent and not part of any actual stream processing system. The late materialization model should be implemented in such a system and then

checked for performance. Also additional capabilities such as choosing between lazy and eager materialization can be tested in this case.

- **Windowed joins:** Our approach in this thesis is based on full history joins. This is primarily because tuple reconstruction requires original tuples to remain in the store in case a reconstruction request for that tuple comes in. For windowed joins, only the tuples that are part of the current window are stored. Extending the model to support windowed joins would allow testing of the effects of late materialization on window based stream join processing.
- **Decoupled reconstruction process:** Separating the reconstruction process from the actual join computation process may yield better results. This will remove the reconstruction overhead from the actual join computation processes.
- **Reconstruction order:** The effect of reconstruction ordering on the run time performance should be investigated. Having relations with different sizes of average payload and then experimenting with different reconstruction orders will prove whether or not using optimized reconstruction yields benefits.
- **Realistic data streams:** Experiments should be carried out using realistic streaming sources such as network packets, location data or Twitter streams. These experiments should be ran over a large period of time e.g., several hours, to see how the storage requirements grow and also how this effects the overall communication costs.

Bibliography

[1] Apache Flume. https://flume.apache.org/.

[2] Apache Kafka. https://kafka.apache.org/.

[3] Apache Kinesis. https://aws.amazon.com/kinesis/.

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

[5] List of countries by number of mobile phones in use. https://en.wikipedia.org/w/index.php?title=List_of_countries_by_number_of_mobile_phones_in_use&oldid=936521540. Accessed: 2020-01-19.

[6] Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Accessed: 2020-01-19.

[7] Powered by Storm. https://storm.apache.org/Powered-By.html. Accessed: 2020-01-19.

[8] TPC-H decision support benchmark. http://www.tpc.org/tpch/.

[9] What happens in an internet minute in 2019? https://www.visualcapitalist.com/what-happens-in-an-internet-minute-in-2019/. Accessed: 2020-01-19.

[10] What is streaming data? https://aws.amazon.com/streaming-data/. Accessed: 2020-01-19.

[11] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980, 2008.

[12] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475, 2007.

[13] C. C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer, 2007.

[14] A. Cuzzocrea. Approximate OLAP query processing over uncertain and imprecise multidimensional data streams. In *Database and Expert Systems Applications - 24th International Conference, DEXA 2013, Prague, Czech Republic, August 26-29, 2013. Proceedings, Part II*, pages 156–173, 2013.

[15] N. Deo. Late-materialization using sort-merge join algorithm. *International Journal of Computer Applications*, 149(2):13–15, Sep 2016.

[16] M. Dossinger, S. Michel, and C. Roudsarabi. CLASH: A high-level abstraction for optimized, multi-way stream joins over apache storm. In *Proceedings of the 2019 International Conference on Management of Data,*

SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pages 1897–1900, 2019.

[17] T. Kolajo, O. Daramola, and A. Adebiyi. Big data stream analysis: a systematic literature review. *J. Big Data*, 6:47, 2019.

[18] T. Kwon, H. G. Kim, M. Kim, and J. H. Son. Amjoin: An advanced join algorithm for multiple data streams using a bit-vector hash table. *IEICE Transactions*, 92-D(7):1429–1434, 2009.

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

[20] K. R. Rastogi. An overview of the join methods in postgresql. `https://severalnines.com/database-blog/overview-join-methods-postgresql`, 2019. Accessed: 2020-01-19.

[21] K. S. Tai, V. Sharan, P. Bailis, and G. Valiant. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 757–772, 2018.

[22] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: producing fast join results on streams through rate-based optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 371–382, 2005.

[23] W. H. Tok and S. Bressan. Progressive and approximate join algorithms on data streams. In *Advanced Query Processing, Volume 1: Issues and Trends*, pages 157–185. 2013.

[24] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[25] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 285–296, 2003.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Kaiserslautern, _____ _____

Date Signature