Near-real time statistics gathered from a continuous and voluminous data mutation stream.

Koen Lavooij

February 1, 2010

Abstract

The amount of digital data is growing fast [1]. Providing that information as a service is not enough, with the amount of information available [2]. To support the users in finding information, supporting systems have been developed to extract specific information from a large amount of stored data.

Finding or extracting interesting information is as least as important as providing the original data. The "collective intelligence" of a large number of users can be used to order the information. The ordered information is of much greater value when compared to the unordered information, because it provides the user with an overview of interesting and less interesting information.

Current database systems are not able to provide ranked information by analyzing a massive amount of user feedback (e.g. clicks) within a short period of time. Therefore, the systems update the answers periodically.

In this thesis, a Stream Processing Engine [3, 4, 5, 6] (SPE) is being adapted. The modified SPE accepts a stream of mutations to a virtual data storage as opposed a stream of tuples. The newly created system exploits the properties of statistical functions in order to efficiently aggregate live statistics over a large stream of mutations.

The newly created system is able to provide answers to a small set of continuous queries. The answers to the queries will be continuously maintained, instead of recalculated. Therefore, the system is able to provide the answers to the continuous queries instantly and with low latency for a large number of users.

Contents

Chapter 1

Introduction

This research concerns a system for the gathering of real-time statistics from a large and volatile data source. Rather than analyze the tuples in the data source, the system will analyze the mutations executed on the data source. An example will be used throughout this thesis to illustrate concepts of the new system.

1.1 Running Example

Consider a news gathering and ranking service. News is very volatile and loses value very fast. Therefore news should be presented to the user as fast as possible. However, the quantity of news and feedback makes it hard to extract news of interest to the user in a timely manner.

The news gathering and ranking system is able to provide a news service in which news is gathered and ranked based on user-feedback (e.g. clicks). The news is ranked by using some mathematical function over the user-feedback.

The news service can use very basic statistics to rank news items, like the number of views on a news item. In order to provide a better list of interesting news items, there is a need to rank news items based upon more elaborate calculations.

A service that can provide ranked news out of statistics should be able to provide the most interesting news:

- according to a statistical query in the system,
- with as little delay possible,
- to a large group of users,
- using a small number of statistical queries,
- by inspecting user-feedback on news of a large group of users.

1.2 Problem Statement

The amount of user-feedback to be examined in order to rate and select the best news may be very large in the news service. Extracting statistics from a large amount of data requires significant computing power.

The system supporting the news service must be able to examine all of the user-feedback and provide news and rankings accordingly. Furthermore, the delay in processing the user-feedback must be within certain time-limits in order to provide ranked news in near real-time.

There is a need for a data management system that is able to continuously process a high volume of mutations to a data-set in order to provide highly accessible result sets of statistical queries.

1.3 Known Problems

A centralized storage results in a system where every query triggers a complete calculation of the query result. That calculation is based upon the tuples in the centralized storage. Systems with a centralized storage are unable to reuse and adjust a query result. Instead, these systems will fully recalculate query results whenever the query is re-executed. In this case every request would trigger recalculation.

Due to the centralized storage used in current data systems, writing and reading from these systems requires access to the same resources [7]. Since simultaneous access to the same tuple in the database can lead to unexpected results, simultaneous access to the same entity is prohibited. However, this reduces the accessibility of the data in such a system [8, 9].

When writing user feedback to the central storage, sections of data need to be locked in order to prevent the use of partially written data in a query result. A second reason for locking when writing, is to guarantee correct storage of tuples. In order to prevent storing two tuples at the same spot (and thereby overwriting one of the tuples), writers should never be able to append tuples to the storage at the same time.

A central storage of tuples would be problematic for the news service. New user-feedback is appended to the storage continuously. When a query is calculated it reads tuples from the storage. If tuples that are used in forming the query result change while calculating the query, the query result is invalid because it does not represent the current state of the database. Therefore, the system locks sections of data that are used in the query result for writing.

The result of locking in databases is that requests to provide a query result set and requests to store user feedback need to wait for each other to finish. The system needs to accept many mutations while simultaneously calculating statistical query results. If the system needs to apply locking to protect its data sources, performance of the system will be severely impeded.

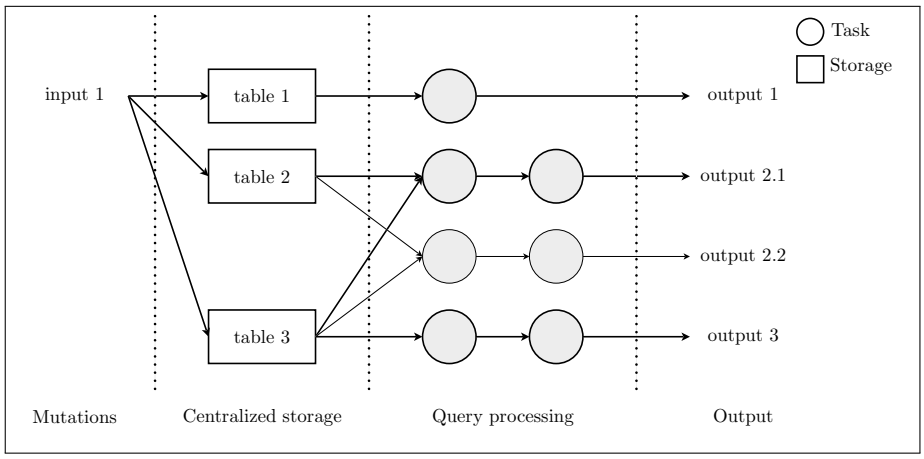

Figure 1.1: RDBMS: post processing

1.4 Current Practices

1.4.1 Relational Database Management System

The use of a Relational Database Management System (RDBMS) [7] to implement the news service is an example of a solution based on a centralized storage. An RDBMS assumes that all original data is stored as tuples in tables and calculates query results using the data stored in tables as depicted in figure 1.1.

When the news service is naively implemented using an out of the box RDBMS, the system performs poorly. Since an RDBMS system is not able to slightly adjust a query result, it starts calculating the query result from the beginning by examining the tuples stored in the tables.

The calculation of queries out of tables can be performed in a very efficient manner by an optimized RDBMS. In order to provide efficient and optimized calculations, every tuple in the data set can be stored in index data structures. Mutations to the dataset and index structures require locks on the entities in order to prevent simultaneous mutations to the same data entity [8, 9]. This limits the number of mutations the system is able to process.

Since the news service is expected to serve a very limited set of queries, the system could be optimized in several ways. For example, a caching system is able to store a query result set in order for the cached result set to be reused in another, similar request. A timeout or a data mutation may invalidate the query results in cache. Requesting an invalidated, cached query result triggers a recalculation of the query result set. A large, volatile stream of data-mutations would invalidate the cached result sets of each query repeatedly and within very short amounts of time.

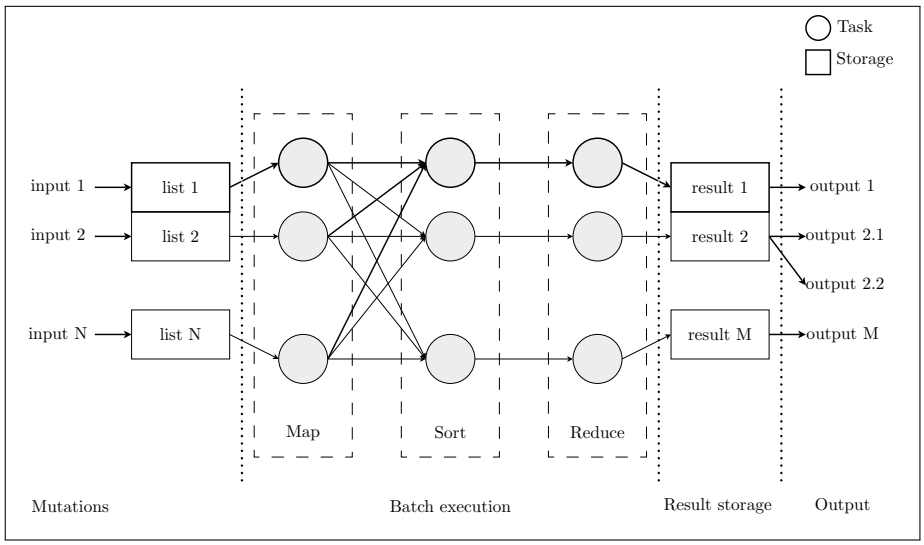

Figure 1.2: Map-reduce: batch processing

1.4.2 Map-reduce

Map-reduce is a paradigm that gained a lot of popularity after Google released a paper stating map-reduce is the basis of their systems [10, 11]. The map-reduce paradigm uses programmable operators to periodically scan the data and (re)calculate query answers. The system is popular for being able to work with vast quantities of data.

A map-reduce engine is batch oriented (figure 1.2). New tuples are simply appended to data files and are left unorganized. When a map-reduce engine is triggered it starts a process that reads the data files. A map-reduce operator is able to scan the file and extract data from the file (map). It then sorts the mapped tuples. The final step is to aggregate (reduce) the tuples in the data file.

The map-reduce paradigm is able to partition a large task into many small tasks. Since the map-reduce process consists of several steps, it is bound to batch calculation. The steps in the calculation process all operate on a data set. The data set in map-reduce can be partitioned into pieces. These pieces can be joined later on in the process in order to produce a result set. This allows a map-reduce engine to create many small tasks out of a much larger task. A map-reduce engine distributes the small tasks over a network and the large task is performed in a massively parallel manner [10, 11, 12].

In a map-reduce system it is not possible to run an ad-hoc query on the system. Since the map-reduce system uses programmable map-reduce operators to calculate the results, it is hard to create such query logic ad hoc.

The data source for a map-reduce operation may be a file generated by user feedback, but may also be output generated by another map-reduce process.

In order to express a query, these map-reduce operators can be chained. The output of a map or reduce task may be reused by other, different map-reduce tasks.

The map-reduce paradigm is interesting; it is able to provide for the database needs of Google. Furthermore, all of the steps in the map-reduce process are easily partitioned into pieces and distributed over a vast network of worker nodes. However, since it is not able to meet the requirements of a near-real time system, it is not applicable to this research.

1.4.3 Stream Processing Engine

Stream Processing Engines (SPEs) [3, 4, 5, 6] are relatively new systems. They differ from the previous mentioned systems in a crucial way. Though all previous mentioned systems may adhere to the same data model, there is no need for a physical representation of the data-entities in an SPE. Contrary to the previously mentioned systems, which require a centralized storage of tuples to calculate query results, the concept of the SPE is to process the incoming data instead of storing the data first. Therefore there is no longer a need for a centralized, optimized storage of tuples.

An SPE is a reactive system. The calculations by an SPE are performed when a tuple is read from the input-stream. An SPE produces results according to the data read from the stream.

SPEs consist of operators which are able to read and produce streams of tuples. One example of such an operator is the Filter operator. The Filter operator can be compared to the WHERE operator in SQL. The Filter operator will only forward tuples which satisfy a certain condition.

There are several types of operators in an SPE, all of which can also be found in the SQL algebra. By chaining operators one can express a query. At the end of such a chain is an operator which produces a stream of mutations that maintains the query result set as seen in figure 1.3. The query result set is stored and is used to process requests.

Some operators require the storage of runtime information to function. Consider an Aggregate operator which counts the views on news articles. The operator produces a stream that describes a set of tuples with an article and a view-count. For every tuple in the incoming stream, the operator needs to find out how the insertion of the tuple affects the view-count of an article. The Aggregate operator has to keep track of the views on the articles in its state in order to find out how a tuple changes the view-count of an article.

Current SPEs are geared towards low latency [19]. In order to ensure low latency the systems are allowed to drop tuples in order to clear the workload [13, 14, 15, 16]. The effect of dropping tuples is that some tuples never make it to the result sets. Current SPEs thereby allow the query result to be an approximation of the actual situation. Furthermore, SPEs are generally used to create an overview of the current situation and tend not to store an extended history of tuples.

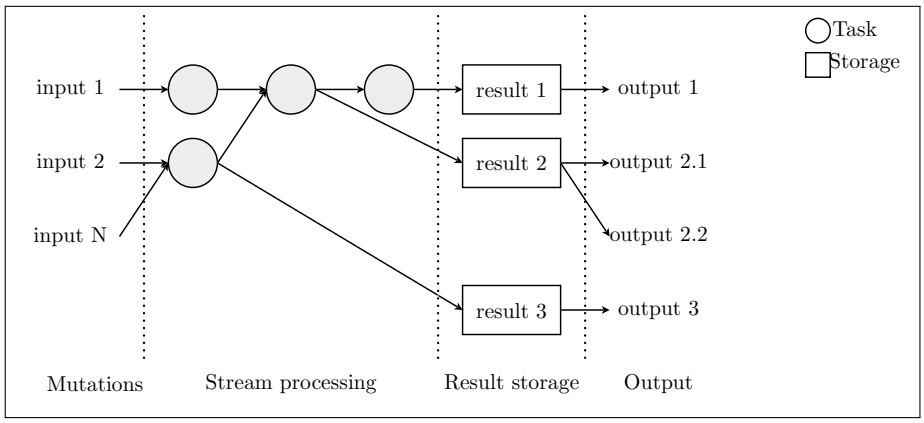

Figure 1.3: SPE: preprocessing

1.5 Research Questions

A system is proposed based on the concept of an SPE. The Analytic Stream Processing Engine (ASPE) has a virtual data storage. Contrary to current SPE design, rather than analyzing tuples streaming into the system, the ASPE analyzes mutations to the virtual data storage. The system is able to use the mutations to adjust statistical query answers. In the ASPE the query results are stored, not the original data.

1. How does one modify a Stream Processing Engine into a statistics query processing tool?
   (a) How to modify a Stream Processing Engine to handle increased state size and produce accurate results?
   (b) What are the strategies in order to optimize the query plan in a Stream Processing Engine?
   (c) In which circumstances is the use of a Stream Processing Engine applicable?

1.6 Overview

The next chapter provides a generic requirements analysis. In the third chapter, a description of the design of the system is written down, after which the solution is validated. After the validation, a conclusion is provided and suggestions are made for future work.

Chapter 2

Requirements Analysis

In section 1.1, some requirements have been mentioned for the news service. From these requirements a list with more generic requirements is derived for the supporting ASPE.

1. The system should be able to provide result sets of statistic queries to the user.
2. The system has to process each mutation inserted into the system in order to provide a verifiable and correct result.
3. The system is able to accept many mutations per second.
4. A mutation in the system needs to be processed within a fixed amount of time.
5. The system should be able to provide query results to a large number of users.
6. The system needs to be able to handle an infinite stream of mutations (insert update delete) on tuples.
7. The system needs to be able to handle an infinite number of tuples described by mutations.

A prerequisite for the service is that the service only needs to provide for a limited set of queries.

2.1 Usage Characteristics

The Analytic Stream Processing Engine (ASPE) does not need to perform ad hoc queries. Instead, the system allows for a few continuous queries. Continuous queries are long running queries and users of the system will mostly reuse these queries rather than change or add queries in the system.

A user of the system sends transactions to the ASPE. These transactions are one of two types: read-transactions and write-transactions. The read-transaction can only consist of one request for a query result set. A write-transaction consists of one or more mutations.

Because the lack of a central storage for tuples, the system is not able to guard against data integrity violations. The system therefore assumes that consistency is not broken by any of the transactions depicted in the input of the system.

The assumption that the user does not break data integrity means the user does not break any of the following rules:

- The user does not insert a tuple with a primary key into a table, if that table contains another tuple with the same primary key.
- The user does not update a tuple that does not exist in the table specified by the user.
- The user does not delete a tuple that does not exist in the table specified by the user.
- The user does not break referential integrity.

The system does not (and cannot) check for these rules to be followed by the user.

2.2 Data Characteristics

When a continuous query is added to the system, the query result set of a continuous query is initially empty. The ASPE will build and maintain the result set of that query by mutations written to the system after the query was added to the system.

The system accepts any mutation to the system from a stream. Since a stream does not necessarily end, the data stream must be able to describe potentially infinite tuples.

The majority of mutations to the tables in the system are assumed to be insert mutations, the inserted tuples will rarely be subject to updates or deletion.

2.3 Scalability

There are some factors that influence the performance of the system. For some of these factors we define the target scalability in order for the system to function and keep functioning in the intended environment.

- The number of mutations inserted into the system per second.
- The number of tuples depicted by the mutations.

- The number of request for result sets per second.
- The number of simultaneous continuous queries.
- The number and complexity of the queries in the system.

2.3.1 Mutations per second

The system should scale linearly over the number of mutations per second inserted into the system per second. Since most of the work in the system is performed when mutations are accepted by the system, it would be ideal to scale well over the number of mutations per second.

2.3.2 Number of Tuples

The system should scale constantly over the number of tuples inserted into the system. The system must be able to accept mutations from an endless stream which only describes insert mutations. Therefore the number of tuples in the system should not influence the speed of operations. This way the input streams of the system can continue submitting tuples without endangering the continuity of the system.

2.3.3 Requests per Second

The number of requests per second should also scale linearly in the system. Although servicing the query result sets requires no post calculation, there is still work to be performed at each request in the form of serialization of tuples.

2.3.4 Number and Complexity of Queries

The scalability in complexity or number of queries is not predictable and is hard to influence. However, since the system is not geared towards ad hoc querying, scalability in these factors is not a priority.

2.4 ACID Properties

ACID is an acronym for Atomicity, Consistency, Isolation and Durability [7]. These ACID properties are properties for a database to ensure the correct handling of transactions.

Consider a situation where the news-service needs to relate article views to metadata concerning that article. In such a data model, tuples depicting a view on an article refer to article metadata in another table. If no such metadata exists, the user has to insert both the metadata and the view at once within one transaction in order not to break referential integrity.

Transactions in the proposed system are a means to allow the system to temporarily put the database in an inconsistent state. Write-transactions in the

system are collections of mutations. While not all write-transaction are closed the ASPE assumes it is in an inconsistent state.

In order to ensure a query result in which referential integrity is not broken, the system prevents the interlaced execution of write-transactions with read-transactions. The non-interlaced execution of write-transactions with read-transactions will ensure consistent query result sets.

Write-transactions may be interlaced with other write transactions. The ASPE expects all actions of the user not to break data-integrity. That also holds for the actions of users within write-transactions. Therefore, the ASPE expects write-transactions not to influence each other. Since isolation would not allow for all transactions to be performed in parallel, isolation of transactions in the proposed system is unwanted.

Atomicity is provided only between read and write-transactions. Atomicity between write-transactions would be an implementation of isolation and is therefore also unwanted.

Durability, for this research, is out of scope. Though it is not very difficult to implement durability, the amount of effort needed to implement durability is significant.

2.5 Assumptions

The system needs to make some assumptions. This section is a summary of these assumptions.

- No write-transaction in the system breaks the integrity rules as depicted in section 2.1.
- Most mutations to the system are insert mutations.

Chapter 3

Design of the Analytic Stream Processing Engine

The ASPE is an adaption of a Stream Processing Engine. The concept of a stream processing engine is modified to allow the system to incrementally adapt result sets of static statistic queries. The ASPE also allows for larger state sizes in operators.

3.1 Solution Ingredients

- The system accepts mutations to a virtual tuple storage. It accepts insert, update and delete mutations.
- The insert and delete mutations messages carry the complete tuple to be inserted or deleted
- The update mutation message carries the complete tuple to be updated as well as the complete tuple to update to.
- Persistency of the tuples depicted in the original stream is not necessary since the system does not need to perform ad hoc queries and does not need to check for mutations breaking data integrity.
- The system does not use tables like an RDBMS, but rather it reads a stream of data mutations (insert, update and delete statements) to a virtual storage and infers tuples from those statements.
- The system consists of operators which produce a stream of mutations based on mutations read from input-streams of mutations.
- The output stream produced by operators maintain the result set of (the half fabricate for) the end-query result.

- By chaining operators into a query tree, one can express a query.
- Heartbeat messages are added to the system to support transactions. They are also used to support garbage collection on the ASPE.
  - Heartbeat messages are synchronized on all streams.
  - Heartbeat messages can not be interleaved with write-transactions.
  - The arrival of a heartbeat message signals that the source stream of that heartbeat signal is in a consistent state. Heartbeats can be configured to occur many times per second.
- The system is layered into a messaging layer, a logic layer and a managing layer.
  - The messaging layer propagates mutation and heartbeat messages through the system. The messages in the messaging layer are able to cross network boundaries in order to give the system the ability to scale up from a single computer to a computer network.
  - The messaging layer delivers the messages from a mutation stream or operator to another operator or query result. These messages are delivered in order.
  - The logic layer is where operators are performing. These are the Join, Aggregate, Filter and Projection operators. All operators are blocking operators.
  - The managing layer is responsible for the layout and distribution of operators on a network of machines. The managing layer is able to optimize and reorganize the layout of operators in order to perform queries in the system more efficiently.

3.2 Solution Details

3.2.1 Transactions

The moment all write-transactions are closed will pinpoint the time where the result sets are assumed to be consistent. In order for a read transaction to read a consistent result, a read-transaction must wait for the data to become consistent.

Write-transactions can not be interlaced with heartbeat messages. When a heartbeat message is read from a stream by an operator, the operator concludes that all write-transactions have closed and that the data-set depicted by the stream is consistent. The system therefore is able to serve consistent results to read-transactions when a heartbeat message arrives.

When all write-transactions are closed the system can provide consistent query results to the pending read-transactions. While servicing read-transactions,

mutations to the query result sets are queued until all read-transactions are closed.

The system is not able to guard against mutations that break consistency. This is because of the absence of a central storage where the system could check for mutations breaking data integrity. The system has to trust users not to break consistency. The proposed transaction system provides the users with a tool to maintain consistency.

3.2.2 Query Algebra

An operator in the system produces a stream of data mutations. The input of an operator consists of one or more streams of data mutations.

The query algebra consists of operators which can be found in the SQL algebra. These are the Filter, Projection, Aggregate and Join operator. Chaining these operators gives the same expressiveness as the SQL language.

Operators are responsible for maintaining a virtual data set with mutations. The resulting data set is derived from another virtual data set. An operator reads mutation messages from its input stream. The mutation messages carry tuples which the operator uses to infer changes to the dataset the operator maintains. Mutations read from the input stream are translated by an operator into mutations to the dataset it maintains. An operator puts these translated mutations on its output stream.

- The Filter operator selects or rejects tuples from the virtual data set.
- The Projection operator is able to project one tuple to another in the virtual data set.
- The Aggregate operator collects tuples by key and extracts statistics out of these tuples and puts the statistics in the virtual data set.
- The Join operator can join tuples from two sources together and put the result in the virtual data set.

Consider a query that counts all views on articles within a specific category as depicted by the article metadata (figure 3.1). The query-tree has two input streams. The first stream maintains the table with article-views, the other stream maintains the table with metadata.

The article-views are aggregated. The output stream of the Aggregate operator now contains mutations that maintain the set of tuples that contain the article-id and view-count. The Aggregate operator maintains the result set of the SQL query

```
1 SELECT articleId, COUNT(*)
2 FROM Views
3 GROUP BY articleId.
```

In the other branch of the tree, metadata is filtered. The input stream of the Filter operator maintains the set of tuples representing all articles and their

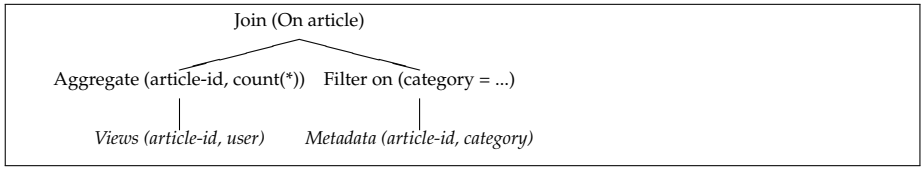

Figure 3.1: Query tree

categories. The Filter operator maintains a set of tuples depicting articles that belong to a specific category. The output stream of the Filter operator contains mutations that maintain tuples in the result set equivalent to the result of the SQL statement

```
1 SELECT articleId, category
2 FROM Metadata
3 WHERE category = '...'.
```

These input streams are then joined together. So that the output of the Join operator maintains the result set that is described by the SQL statement

```
1  SELECT articleId, count
2  FROM
3      (
4          SELECT articleId, COUNT(*) AS count
5          FROM Views
6          GROUP BY articleId
7      ) AS viewsPerArticle,
8      (
9          SELECT articleId, category
10         FROM Metadata
11         WHERE category = '...'
12     ) AS categoryArticles
13 WHERE
14     categoryArticles.articleId = viewsPerArticle.articleId
```

which is equivalent to the SQL statement

```
1 SELECT articleId, COUNT(*)
2 FROM Metadata, Views
3 WHERE
4     Metadata.articleId = Views.articleId
5 AND
6     category = '...'
```

Filter

Operators produce a stream of mutations that maintain a set of tuples derived from an input-stream of mutations. In the case of the Filter operator, the operator would forward, modify or ignore mutation messages. The Filter operator maintains a subset of tuples depicted by its incoming stream (figure 3.2).

An insert mutation message contains the tuple to be inserted. If the tuple is allowed in the result set of the Filter operator, the message is forwarded to the

Incoming mutation	Condition	Outgoing mutation
Insert $t_{inserted}$	accepted($t_{inserted}$) ¬accepted($t_{inserted}$)	Insert($t_{inserted}$) -
Update t_{old} to t_{new}	accepted(t_{old}) ∧ accepted(t_{new}) accepted(t_{old}) ∧¬accepted(t_{new}) ¬accepted(t_{old}) ∧ accepted(t_{new}) ¬accepted(t_{old}) ∧¬accepted(t_{new})	Update(t_{old} to t_{new}) Delete(t_{old}) Insert(t_{new}) -
Delete $t_{deleted}$	accepted($t_{deleted}$) ¬accepted($t_{deleted}$)	Delete($t_{deleted}$) -

Figure 3.2: Overview of the logic in a Filter operator

output stream. If the tuple is not allowed in the result set, the insert mutation is ignored.

An update mutation message contains the tuple to be updated (t_{old}) and the updated tuple itself (t_{new}). If both t_{old} and t_{new} are allowed in the result set of the Filter operator, the update mutation is propagated to the output stream.

If t_{new} is allowed in the result set, but t_{old} was not, an insert mutation is placed on the output stream, inserting t_{new}. Since t_{old} was never propagated to the output stream by this operator, the tuple will be ignored.

If t_{new} is not allowed, but t_{old} was, the result would be a delete mutation, deleting tuple t_{old} from the result set. If both the old and the new tuples are not allowed in the result set, the update mutation is ignored.

Finally, a delete mutation message is propagated if its tuple is in the result set, otherwise the mutation is ignored.

Projection

The Projection operator is capable of projecting tuple $t_{original}$ onto a new tuple $t_{projected}$ using a projection function f. The projection function is a deterministic function that accepts one tuple as a parameter and returns one tuple. The tuples depicted in the insert, update and delete mutations all are transformed using that same projection function. The mutation messages are then placed on the output stream (figure 3.3).

Incoming mutation	Outgoing mutation
Insert $t_{inserted}$	Insert(f($t_{inserted}$))
Update t_{old} to t_{new}	Update(f(t_{old}) to f(t_{new}))
Delete $t_{deleted}$	Delete(f($t_{deleted}$))

Figure 3.3: Overview of the logic in a Projection operator

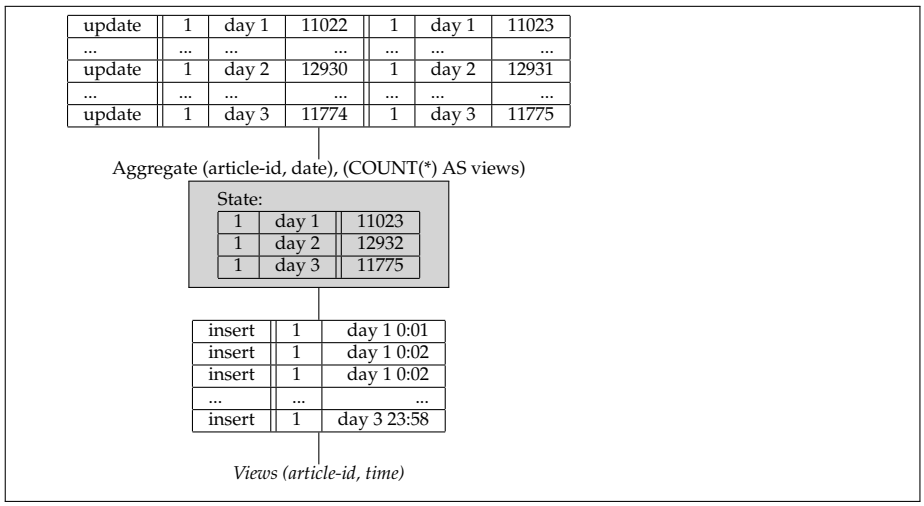

Figure 3.4: Aggregate operator with output, state and input

Aggregate

The Aggregate operator maintains summaries of groups of tuples. These summaries are called aggregates. The operator extracts a key from the tuples depicted by an input stream in order to determine the aggregate a tuple belongs to (figure 3.5). The output stream of the Aggregate operator consists of mutations that maintain the summaries of the groups extracted from the tuples depicted in its input stream. In other words: the output stream consists of mutations to all the aggregates the Aggregate operator maintains.

For instance, an Aggregate operator can count the number of views on articles. The input stream for that Aggregate operator consists of tuples containing an article-id. Each tuple in the input stream represents a view on an article. The Aggregate operator uses the article-id as a key and maintains a count of tuples with that article-id. These two values are stored in an aggregate. The number of views per article per day can easily be maintained without storing the original tuples in the input stream (figure 3.4).

The Aggregate operator is able to extend an aggregate by means of aggregation-functions like *mult* or *sum*. These functions combine the last emitted aggregate with a mutation depicted in the input stream to form and emit a new aggregate. Aggregation-functions therefore incrementally adjust an aggregate rather than using all known tuples in the group in order to recalculate an aggregate. The aggregation-functions are programmable.

Functions like *mult* and *sum* are associative and commutative. Aggregation-functions with these properties can operate without a state to store parts of the input-tuples into. The associative property states that it does not matter how the numbers are grouped, the operator will still deliver the same result $(1 + (2 + 3) = (1 + 2) + 3)$. The commutative property $(1 + 2 = 2 + 1)$ of a

Incoming mutation	Condition	Outgoing mutation
Insert $t_{inserted}$	hasAggregate(key($t_{inserted}$))	Update(aggregateOf($t_{inserted}$) to aggregateOf($t_{inserted}$) + $t_{inserted}$)
	¬hasAggregate(key($t_{inserted}$))	Insert(newAggregate($t_{inserted}$))
Update t_{old} to t_{new}	aggregateOf(t_{old}) = aggregateOf(t_{new})	Update(aggregateOf(t_{old}) to aggregateOf(t_{old}) - t_{old} + t_{new})
	aggregateOf(t_{old}) $\neq$ aggregateOf(t_{new})	*Handle as* *Incoming delete(t_{old})*, *Incoming insert(t_{new})*
Delete $t_{deleted}$	count(aggregateOf($t_{deleted}$)) > 1	Update(aggregateOf($t_{deleted}$) to aggregateOf($t_{deleted}$) - $t_{deleted}$)
	count(aggregateOf($t_{deleted}$)) = 1	Delete(aggregateOf($t_{deleted}$))

Figure 3.5: Overview of the logic in an Aggregate operator

function means order is not important for the outcome of the function. Many statistical functions can be calculated using associative and commutative operations: *Min*, *Max*, *Sum*, *Count*, *Average*, *Standard Deviation*. Current cannot adjust query results and maintain windows of tuples. Using the tuples in the window, the SPEs calculate the aggregates [17].

Aggregation-functions need to store more data when the function is not associative and commutative. For instance an aggregation-function that calculates how many unique values are inserted needs to store a list of values in order to determine if a value is unique or not. It also stores per value a number of occurrences in order to determine whether that value still exists in that group after an update- or delete-mutation.

Given an insert mutation, the Aggregate operator extracts the key-values from the tuple. These key values depict the aggregate the tuple belongs to. The Aggregate operator retains the last emitted aggregates in its state and incrementally combines the aggregate with the newly inserted tuple in order to determine the new values of the aggregate.

If there is no aggregate-tuple with the key depicted by the inserted tuple,

a default, empty aggregate-tuple is created. After the default aggregate-tuple has been created, the group is incrementally adapted using the tuple provided with the insert mutation message.

At the arrival of update-mutations the Aggregate operator inspects whether the updated tuple still belongs to the aggregate depicted by the original tuple. If the tuple belongs to another aggregate due to the update, the system deletes the original tuple from its current aggregate and appends the updated tuple to its new aggregate. Since this update would affect two aggregates, the effect is that the Aggregate operator needs to send two mutation messages onto its output stream.

If the aggregate is $\langle article_1, 10 \rangle$ and tuple $\langle article_1 \rangle$ is appended to the aggregate-tuple, the operator sends a mutation message to update $\langle article_1, 10 \rangle$ to $\langle article_1, 11 \rangle$. The Aggregate operator then stores $\langle article_1, 11 \rangle$ as the current aggregate in the operator state.

Delete mutations lower the count in the aggregate. If this mutation reduces the aggregate-size to zero, a mutation is put on the output stream that deletes the aggregate from the result set.

Join

The Join operator combines the tuples of two sources based on a common key (figure 3.7). Every tuple depicted by its input streams is stored in hash tables by that key. Each input stream has its own hash table to store tuples in (figure 3.6).

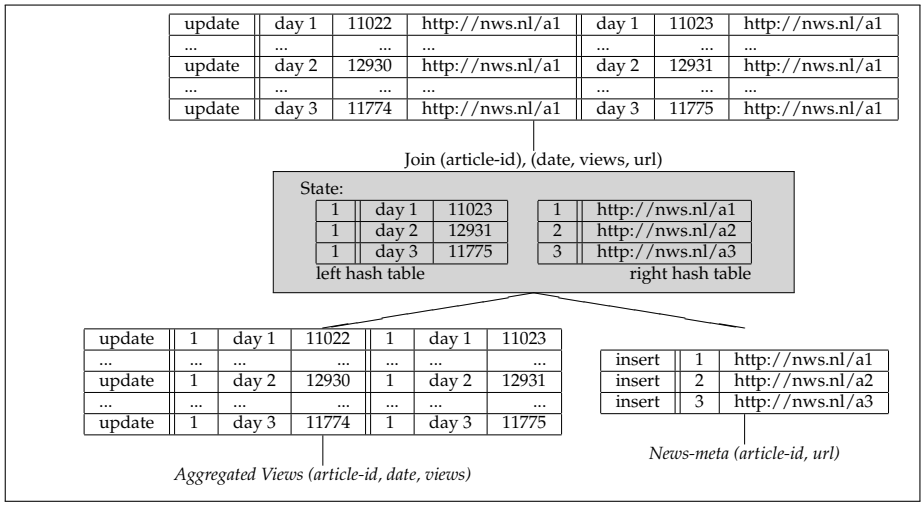

Figure 3.6: Join operator with output, state and input

Given an insert mutation with tuple $t_{inserted}$, the Join operator extracts the key from the tuple. The Join operator queries the hash table of the other stream for tuples with a matching key ($j_1..j_n$). The inserted tuple is glued to the match-

ing tuples of the other input stream. The glued tuples $t_{inserted}.j_1$ .. $t_{inserted}.j_n$ are inserted into the result set.

When an update mutation is received depicting an update from t_{old} to t_{new}, the Join operator first checks whether the join-key is affected by the update. If the join-key is not affected, the matching tuples $j_1..j_n$ are queried from the hash table of the tuples of the other stream. The operator then propagates the update $t_{old}.j_1$ to $t_{new}.j_1$ .. $t_{old}.j_n$ to $t_{new}.j_n$.

If the join-key is affected, two sets of matching tuples $j_1..j_n$ (using the key from t_{old}) and $j2_1..j2_n$ (using the key from t_{new}) are looked up in the hash table of the tuples of the other stream. The operator then propagates the deletion of $t_{old}.j_1..t_{old}.j_n$ and the insertion of $t_{new}.j_1$ .. $t_{new}.j2_n$.

On the arrival of a delete mutation of tuple $t_{deleted}$, the join-key is extracted from the tuple. The Join operator queries the hash table of the other stream for tuples with a matching join-key ($j_1..j_n$). The deleted tuple is glued to the matching tuples of the other input stream. The glued tuples $t_{deleted}.j_1$ .. $t_{deleted}.j_n$ are then removed from the result set.

Naturally all the mutations depicted in the input stream will also be reflected in the hash tables containing the tuples depicted by the stream.

3.2.3 State Space

In order for operators to produce a mutation stream, some operators need to maintain a state. The Aggregate operator needs to store tuples of the aggregated groups. The Join operator has to store all tuples from multiple sources in hash tables in order to be able to lookup and join with these tuples. Therefore the state space of the Join operator is inclined to grow faster than the Aggregate operator.

Because the stream of data mutations for every operator is to be considered endless, the state space of all operators might grow indefinitely if not limited somehow.

Garbage Handling and Collection

A strategy needs to be devised that is able to remove tuples from the state spaces.

Whenever tuples are removed from the state spaces, mutations on these tuples are obviously no longer supported by the operator. Mutations in the input stream of an operator regarding tuples removed from the state are ignored. Therefore mutations to the removed tuples are no longer reflected in the result sets of queries.

When a state space is cleaned, the removal of the tuples is not propagated to subsequent operators as a delete-mutation and the result set of the operator in question does not change. Therefore, data that is removed from the state space of an operator becomes immutable but is still present in the result set of that operator. The data removed from the state is garbage and is completely removed from the system.

Incoming mutation	Condition	Outgoing mutation
Insert $t_{insleft}$	-	{Insert($t_{insleft} \bowtie x$) | $x \in probehash_{right}(t_{insleft})$}
Insert $t_{insright}$	-	{Insert($t_{insright} \bowtie x$) | $x \in probehash_{left}(t_{insright})$}
Update $t_{oldleft}$ to $t_{newleft}$	key($t_{oldleft}$) = key($t_{newleft}$)	{Update($t_{oldleft} \bowtie x$ to $t_{newleft} \bowtie$ x) | $x \in probehash_{right}(t_{oldleft})$}
	key($t_{oldleft}$) $\neq$ key($t_{newleft}$)	{Delete($t_{oldleft} \bowtie x$) | $x \in probehash_{right}(t_{oldleft})$} {Insert($t_{newleft} \bowtie x$) | $x \in probehash_{right}(t_{oldleft})$}
Update $t_{oldright}$ to $t_{newright}$	key($t_{oldright}$) = key($t_{newright}$)	{Update($t_{oldright} \bowtie x$ to $t_{newright} \bowtie$ x) | $x \in probehash_{left}(t_{oldright})$}
	key($t_{oldright}$) $\neq$ key($t_{newright}$)	{Delete($t_{oldright} \bowtie x$) | $x \in probehash_{left}(t_{oldright})$} {Insert($t_{newright} \bowtie x$) | $x \in probehash_{left}(t_{oldright})$}
Delete $t_{delleft}$	-	{Delete ($t_{delleft} \bowtie x$) | x $\in probehash_{right}(t_{delleft})$}
Delete $t_{delright}$	-	{Delete($t_{delright} \bowtie x$) | x $\in probehash_{left}(t_{delright})$}

$probehash_{table}(tuple)$ = {x \| x ∈ table \| keyOf($tuple$) = keyOf(x)}

Figure 3.7: Overview of the logic in a Join operator

For instance when applying this method to the query tree depicted in figure 3.1, tuples can be deleted from the state space of the Aggregate operator. But when the deletion from the state-space is not propagated to the output stream of the operator, the result-set of the Aggregate is not modified (figure 3.8). Since the output of the result-set of the operator is used by the join operator, the result of the Join operator is not modified.

Aggregate operators have aggregates that become immutable when the aggregates are removed from the state of the operator. Since the deletion of an aggregate is not propagated to subsequent operators, the aggregate is still represented in the result set of the Aggregate operators.

When the state of Join operators is cleaned by the garbage collector, no tuples are deleted from the result set of the Join operator. Once tuples depicted by an input stream are cleaned from the state of a Join operator, the operator is no longer able to join to these tuples. Due to this effect, the system is unable to join to data that has become immutable.

In the query tree depicted in figure 3.1 we could remove metadata tuples

from the state space of the Join operator. Since this change is not propagated, the result set of the query still consists of tuples joined with the removed metadata tuples. However it is no longer possible to join to these tuples. Therefore views on articles with removed metadata tuples are no longer counted or propagated to the result of the query.

Selecting Which Data to Make Immutable

Strategies can now be defined to select data that will become immutable.

Garbage collection in the system is performed when a tuple selection strategy is applied to a state space in order to collect the "garbage" in the state space.

Statistics are usually gathered in a dimension of time; old information (gradually) decreases in value [18]. That notion has been used to select which data should become immutable. For each table in the system an interval after which data becomes immutable is selected. Per query result data is aggregated into timeslots with a fixed granularity (e.g. days or months). This principle is used to specify strategies for the removal of tuples in the state spaces of t for data that is aggregated per day and becomes immutable after 24 hours, the system only needs to maintain 2 time slots of one day (figure 3.8).

For an Aggregate operator this means that the key columns to which the tuples are grouped is extended with a field indicating the timeslot. The tuples of aggregate-tuples that are outside of the mutability window are deleted from the state space.

The garbage collection principle also applies to the Join operator. Though Join operators might have a different garbage collection strategy for the tuples of each input stream. All tuples in a hash table of the Join operator must contain a value depicting the time slot in order for the garbage detection system to determine whether the tuple is in or out of the mutability window. The garbage detector iterates over all tuples in the state space in order to delete tuples out of the mutability window.

3.2.4 Query Optimization

By chaining operators one can express a query. Optimization strategies for the ASPE have similarities to optimizations for RDBMS systems. In order to create a more efficient query, the order of the operators may need to be switched. Because of long running queries in the ASPE system, sub-queries in this system can be shared between queries thereby reducing the overall state size of the system.

Creating an efficient network of operators is essential. The efficiency of the system can be expressed in the number of mutation messages and state size. In this section the optimizations specific to the ASPE are mentioned.

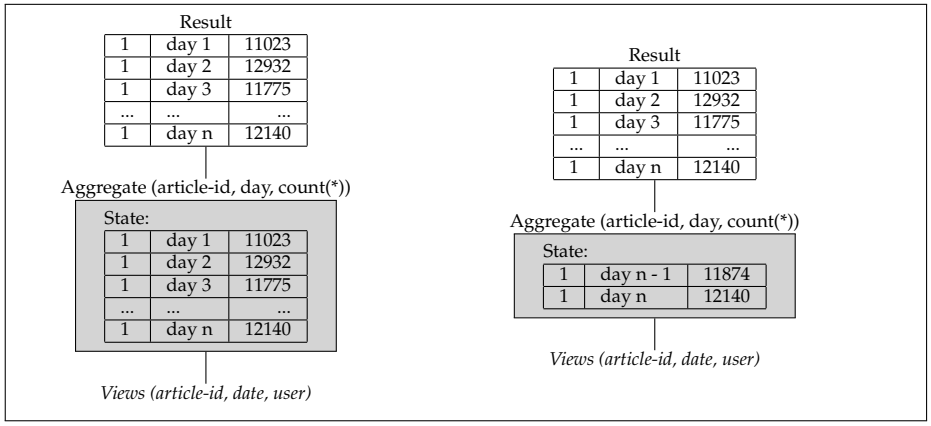

Figure 3.8: Garbage collection: before and after

Operator Reuse

Many queries share a part of their query tree with another query in the system. The system tries to reuse common query trees. This makes the system more efficient. For instance, consider the query tree in figure 3.1. We could make two instances of query trees like the tree in figure 3.1, both with different categories. The left part of the query tree responsible for counting the views on articles can be reused in both query trees.

Maintaining Flow

In known SPEs the flow is maintained by ignoring mutations that cannot be processed in time. Since the ASPE is designed to generate accurate results, the system cannot ignore mutations. Instead, it uses blocking operators. The system is allowed to have some delay in the results from time to time in order for the system to process every mutation.

The number of tuples flowing out of the operator divided by the number of tuples flowing into the operator is called the "tuple forward rate" . The tuple forward rate is critical for optimizing the flow of an operator network. The less tuples in the system, the better the flow.

If the operators in a query tree can be reordered, the system prioritizes the operators with the lowest tuple forward rates before operators with higher tuple forward rates in the operator sequence. Placing the operators with the lowest tuple forward rates up front reduces the number of overall mutations and the number of overall tuples in the system.

Minimizing the Need for State Space

Join operators maintain a state space in which all the mutable tuples of both input streams are stored. In contrast Aggregate operators only maintain a re-

duced set of tuples that represent aggregate-tuples in the state space.

By rewriting the query tree to sequence Aggregate operators before Join operators, the overall number of tuples stored in the state space of the total query tree is substantially reduced.

Consider a query that counts the number of views there are on news feeds and compares that to the number of people registered to these news feeds. In this query, data about registrations needs to be joined with data about feeds. Data needs to be aggregated per article in order to determine the numbers of views and registrations.

In the example in figure 3.9 the "Views" mutation input-stream is given a rate of 0.99 and the "Registrations" mutation input-stream has a rate of 0.01 which depict their relative volumes of mutations in the system. Assumed is the worst case memory complexity for a tuple stream; the tuple streams only consist of insert mutations. The average number of views per feed is 100 and the average number of users registered per feed is 100.

The unoptimized query tree in figure 3.9 has a Join operator which stores all tuples. The Join operator itself has a ratio of 1 to 100, which means that for every inserted tuples it averages to join with 100 tuples of the other source. The Aggregate operator aggregates these in a 1 to 10000 ratio. Which means that for every 10000 (100 x 100) tuples inserted, the operator produces 1 tuple.

The total memory complexity of the unoptimized tree is 1.01. That is, it stores 101% of all tuples inserted into the system. The Join operator stores all tuples inserted into the system, which gives the query tree a memory complexity of at least 1.00. Views and registered users are joined together by feed, that Join operation produces an average of 100 tuples per inserted tuple, giving the Join operator a tuple-forward-rate of 100. The Aggregate operator reduces an average of 10000 tuples to 1 tuple. Therefore the Aggregate operator stores 1% of the total number of tuples into the system.

When the Aggregate operator is prioritized before the Join operator figure 3.10) the memory complexity is reduced to 0.02; The system maintains a number of tuples that is 2% of the number of tuples inserted into the system. When views are aggregated in a 100 to 1 ratio, the number of tuples stored by the Aggregate operator is 0.99% of the total number of tuples inserted into the system. The number of tuples stored in the Aggregate operator that aggregates registrations is 0.01% since the input only consists of 1% of the total number of tuples inserted into the system. The two aggregates together store 1% of the total number of tuples in the system.

The Join operator in the optimized query tree can only produce 1 output tuple per 2 inserted tuples, since the aggregates produce 1 tuple per feed, and the Join operator uses that feed to join tuples. The Join operator only has to store the tuples the Aggregate operators produce, and therefore stores as many tuples as the Aggregate operators combined. Therefore the total number of tuples stored in the optimized system is 2% of the total number of tuples inserted into the system.

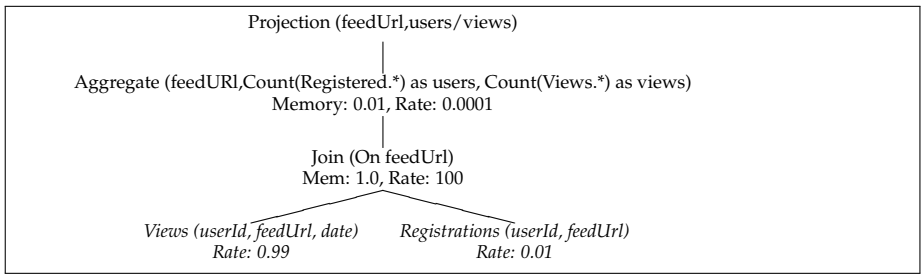

Figure 3.9: Query tree extended with rates and memory complexity

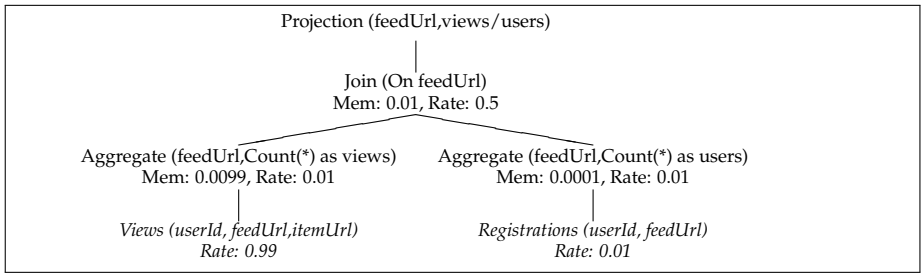

Figure 3.10: Optimized query tree

3.2.5 Operator State Space Overflow

As described in section 3.2.3, garbage collection is designed to limit the size of the state space of query operators. Garbage collection deletes every tuple that falls outside of the mutation-window.

There is a chance that this garbage collection scheme is not sufficient. The tuples may need a long time to become immutable. Another reason for operator state space overflow may be a too voluminous input stream.

To counter the overflow of operator state space, the ASPE is able to partition an operator input stream into multiple sub-streams. The operators working on the sub-streams are then distributed over a network of machines. The sub-streams maintain a distinct subset of the tuples depicted in the original stream. The technique involves hashing of key values extracted from the tuples in the input stream. The sub-streams each maintain a set of tuples with a distinct range of hash values.

The stream of the individual operators is collected in order to create one single mutation stream, to be used in operators sequenced in the query tree.

Aggregate operators hash the aggregate-tuple-key as seen in figure 3.11, while the Join operator hashes the fields on which the two streams join (figure 3.12).

Since the assumption was made that most data mutations are data insertions, assumed is that the number of tuples in the system is roughly the number of mutations in the system. Therefore, optimizing for flow or memory requires

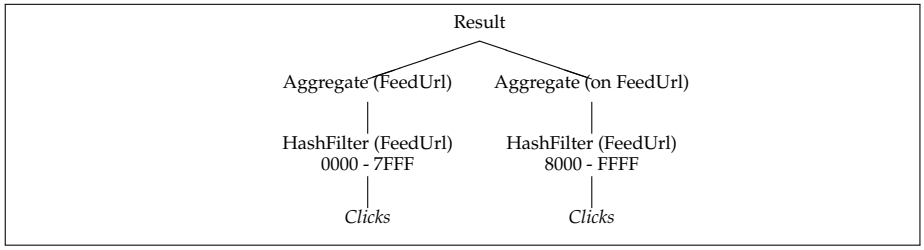

Figure 3.11: Aggregate Query tree with Hash partitioning

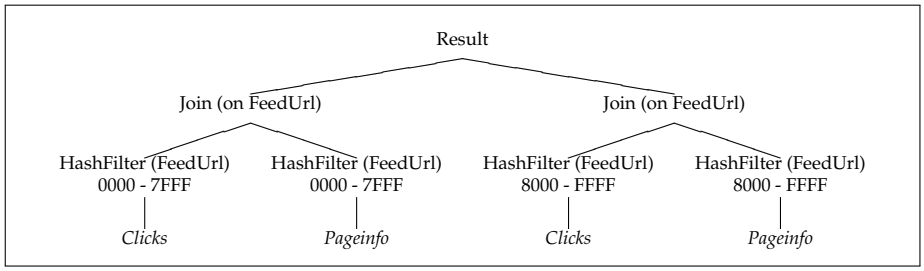

Figure 3.12: Join Query tree with Hash partitioning

the same principles:

1. Prioritize operators with lower rating to be higher in the query tree.
2. Decrease the rating of a stream as much as possible before the occurrence of operators with states (Join, Aggregate) in the query tree.
3. Prioritize Aggregate operators before the more expensive Join operators.

Chapter 4

Solution Validation

In this chapter the solution described in the previous chapter is validated. The solution validation is divided in two parts. First a qualitative validation is performed in which a theoretical approach to validation is applied. Secondly a quantitative validation is given.

4.1 Qualitative Validation

In this section a the design of the ASPE validated using the requirements specified in chapter 2. The design specifications in chapter 3 are used to assess the quality of the proposed system.

4.1.1 Scalability

A qualitative analysis is provided based on the scalability parameters specified in section 2.3.

Mutations per Second

The Filter and Projection operators scale linearly over the amount of mutations per second. This is because both the Filter and the Project operator do not maintain a state space.

When state spaces are involved, the ASPE cannot guarantee linear scalability over the number of mutations per second. For example, the number of aggregates in the state of an Aggregate operator may vary from one to many aggregates. With more aggregates in the state, lookup-time of aggregates will be longer.

Join operators are even more sensitive to varying lookup-times, since all tuples inserted by the sources of Join operators are stored in the state space of the Join operator. Furthermore a single mutation to a Join operator can have an impact on many emitted tuples thereby producing a lot of mutation messages.

It is possible to prevent Join operators to be placed in situations where the Join operator is able to join one tuple to many other tuples. When Join operators are well placed, the amount of work per mutation on the system input stream is very close to constant.

Number of Tuples

For the Projection or Filter operator the number of tuples does not have any effect on the speed of operation or the memory usage of the operator. However, both the Aggregate operator and the Join operator store tuples in order to function.

The ASPE is able to maintain an aggregation of tuples, without storing the set of tuples which make the aggregate. This allows the system to suffice by storing the aggregate-tuples and incrementally adapt aggregate-tuples according to the mutations in the input stream. Therefore an Aggregate operator only needs to store a fraction of the number of tuples depicted by its input stream of mutations.

The Join operator maintains a history of the tuples described by incoming mutation streams. This enables the Join operator to combine tuples from two input streams. Since the Join operator stores many tuples, it is the most susceptible to a large amount of tuples described by its input streams.

Requests per Second

The ASPE does not need to perform any form of post processing when servicing a query result set except for serializing the tuples in the result set. The number of requests served per second therefore scales linearly.

Number and Complexity of Queries

When a query shares part of its query tree with another query, the subtree of that query is not added to the system twice. Instead, the system connects both query trees to the shared subtree. This gives the system the capability to scale better over the number and complexity of queries.

Garbage Collection

The garbage collector deletes tuples from state spaces, making space for new tuples to be inserted. Cutting down the size of state spaces also shortens the lookup of tuples in the state space. Garbage collection therefore maximizes the number of tuples the ASPE can process. Garbage collection also minimizes the effect of a large number of tuples on the number of mutations the ASPE can process per second.

4.1.2 ACID Properties

The ASPE therefore supports relaxed ACID properties of transactions. By preventing read-transactions to be executed until all running write-transactions are finished, the system is able to provide a consistent query result.

Since the system assumes that the user does not break data-integrity, there is no need for Isolation or Atomicity. Isolation and Atomicity of write-transactions are unwanted properties since both of these decrease accessibility of the ASPE.

Durability can be achieved by keeping a log of the mutations to the virtual storage of the system.

4.2 Quantitative Validation

In this section a quantitative validation of the ASPE is performed. First a test is performed using well known RDBMS solutions in order to provide insight into the complexities involved with inserting tuples and querying RDBMS systems.

Secondly, a test is conducted in order to find out how the ASPE performs compared to known RDBMS solutions. Finally, a test will be performed to examine the behavior of the ASPE when reading from a large stream of data.

4.2.1 Data

AOL published queries to its search engine of about 650,000 users over three months [20, 21]. We will use this dataset for the experiments.

For most of the tests only a portion of this dataset is used, due to time constraints. The total dataset covers a month of searches, stored as 2.2 GB of raw text data. The dataset consists of more than 35 million tuples.

The dataset consists of tuples containing five values:

- *user*
  The tuples contain an anonimized user id.
- *query*
  The query entered by the user into the search engine.
- *time*
  The time at which the query is performed on the search engine.
- *URL*
  The URL selected from the items provided by the search engine
- *rank*
  The rank of the selected url in the items provided by the search engine.

One single table is maintained containing these 5 fields. In order to optimize for some of the queries in the system, the *user* field is indexed. This provides some insight into the influence indexes have on the execution speed of the RDBMS systems.

4.2.2 Test Setup

The tests are performed on three different systems. The first number of tests are conducted to provide insight into how RDBMS systems respond to a large amount of data. Two well-known RDBMS systems are used for this test. The first is MySQL and the second is PostGreSQL. Subsequently, these RDBMS systems are compared to the ASPE.

Four different queries are used in the test. Ranging from simple problems to harder problems, increasing the amount of work to be performed in forming the result sets of the queries. These queries are listed below.

1. How many unique users are known?
   This is an aggregate statement with one result row. The user table is indexed so this query may be optimized by the RDBMS system.
2. What is the average user click-through rate per page request?
   This is a layered aggregate possibly using an index in SQL.
3. How many unique queries are performed per user?
   This is an aggregate possibly using an index in SQL.
4. How many unique URLs are clicked per query?
   This is an aggregate without SQL index optimizations.

4.2.3 Complexities Involved

In some situations MySQL can use indexes to form a query answer. MySQL has counters on its index structures. The amount of work performed in counting (unique) values that are stored in an index is less than counting values in non-indexed fields.

Query 1 counts the distinct users in the system. MySQL optimizes by scanning the index rather than all values in the table. PostGreSQL is not able to use an index optimization in this case. PostGreSQL does a sequential scan over the table in order to form the query result.

For solving query 2 the actions of users are examined. The average click-through rate is calculated from a list of click-through rates grouped per user. PostGreSQL solves this problem through an index scan to extract the tuples concerning individual users. It extracts a list average click-through rates per user. The list is used to create an average click-through rate.

MySQL solves query 2 by first sorting the list of tuples by user. The tuples are then aggregated per user to produce a click-through rate per user. The aggregated result is then further aggregated to extract an average click through rate.

Query 3 and query 4 are solved the same way. MySQL can solve these queries in one sequential scan and direct aggregation. Surprisingly MySQL did not use index counters to solve this query. PostGreSQL solves these queries with a sequential scan, followed by an aggregation step. The difference in the queries is in the amount of unique data to count.

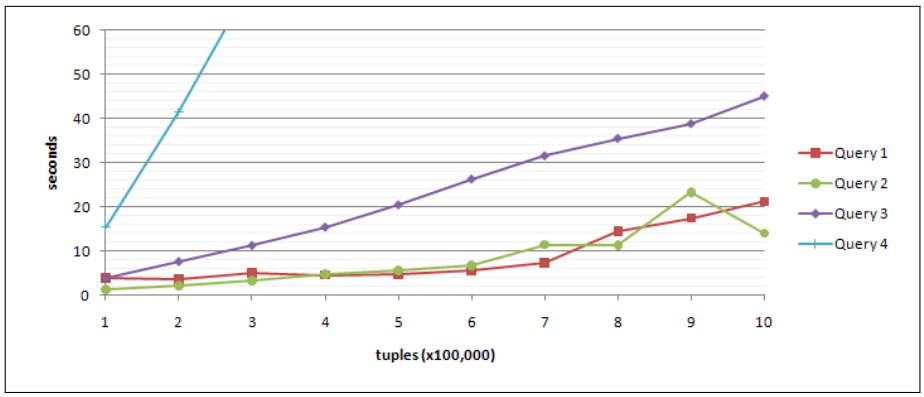

Figure 4.1: PostGreSQL query response time in seconds

4.2.4 RDBMS Response Times

RDBMS systems have a central storage of tuples and therefore append inserted tuples to central storage. Because of an index on the user field of the table, the RDBMS systems need to maintain the index according to the inserted tuples.

Querying an RDBMS with a statistical query triggers the system to examine the indexes and tuples in the central storage in order to build the query result.

In order to gain insight into the effects of a large number of tuples in the central storage, an examination into the performance of RDBMS systems is conducted. Points of interest are the response times of statistical queries and the capability of the RDBMS to accept new tuples.

A test is performed in which 1,000,000 tuples are inserted into the database systems in steps of 100,000. After the insertion of each set of 100,000 tuples, the time it took inserting the tuples is measured. After the insertion of tuples the system is queried. The queries are performed in sequence rather than in parallel in order to provide insight into the individual performance of each query. Performance is measured in response time in seconds.

Figure 4.1 and figure 4.2 show that the number of mutations accepted per second by an RDBMS systems is constant. Both PostGreSQL and MySQL are able to accept the insertion of a steady number of tuples per second. Furthermore, when statistical queries are executed in the RDBMS systems, a reasonably steady number of tuples per second is examined per query. Query response times of RDBMS systems scale linearly over the number of tuples in the central storage.

Since MySQL has counters on its index structures, the amount of work performed in counting (unique) values that are stored in an index is less than counting values in non-indexed fields. Though the index optimization does increase the number of tuples examined per second, the number of tuples in the central storage that is examined per second is still reasonably steady.

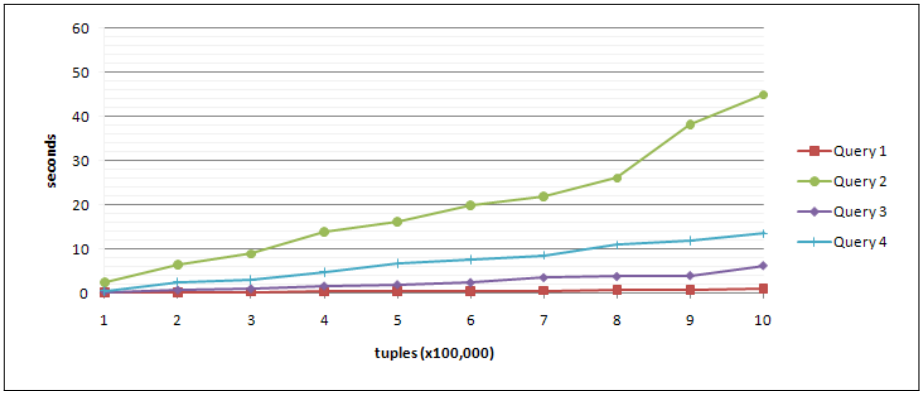

Figure 4.2: MySQL query response time in seconds

4.2.5 RDBMS versus ASPE

In the next test, the PostGreSQL, MySQL and the ASPE are tested in a production environment. Therefore, the test will be conducted in a slightly different way. Tuples still are inserted by steps of 100,000 tuples until 1,000,000 tuples are inserted. After the insertion of the tuples, the queries will be performed in parallel by the RDBMS systems. Performance is measured in seconds from the time the queries are posted until the last query result is received.

Because the ASPE adapts each of the query results on each mutation, the amount of work involved per mutation is higher. However the overhead is near constant. Just like the RDBMS systems, the ASPE is able to accept mutations at a high rate (figure 4.3).

In figure 4.4 query response times of PostGreSQL, MySQL and the ASPE are plotted in one graph. Since the RDBMS systems continue to increase query response times when more tuples are added, it is clear that PostGreSQL and MySQL are not capable of accepting an endless stream of tuples. The systems are also not capable of returning a query result timely. The ASPE system is able to return up to date query answers instantly, but RDBMS system are only capable of refreshing query results on interval, in this case every 100,000 tuples.

Because query result sets of the continuous queries entered into the ASPE are constantly being maintained, querying the ASPE will always yield an instant response. Query response times of the ASPE do not depend on the number of tuples in the database.

4.2.6 ASPE and Large Streams

In order to test the ASPE with a larger number of tuples, the total AOL data set is used. The data set is sorted on the time of entry and fed to the input of the ASPE. A mutability window needs to be introduced to keep the memory size of the ASPE within limits using garbage collection. The mutability window is

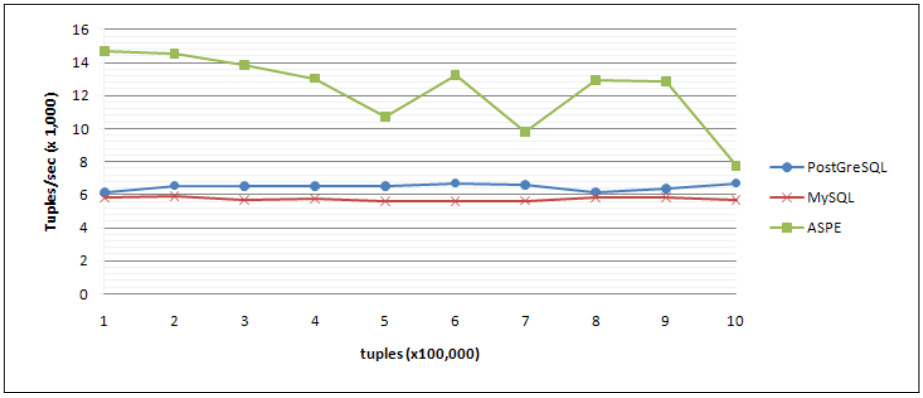

Figure 4.3: Tuple insertion performance

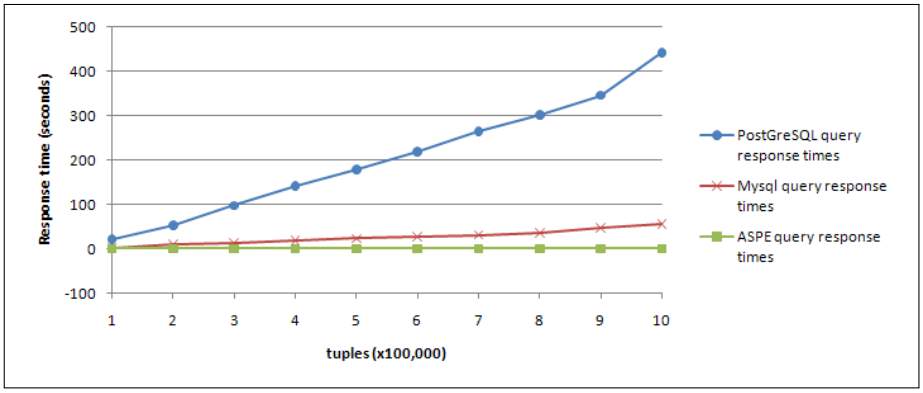

Figure 4.4: Query response times

set to 24 hours.

If garbage collection is not applied, the ASPE is faster but reaches its memory limit before the end of the dataset has been reached. Without garbage collection the number of tuples in state spaces increase and the time it takes the system to perform lookups increases dramatically until the system runs out of memory. When the memory limit of the ASPE is reached, the system is not able to continue (figure 4.5).

When we apply a mutability window of 24 hours to the table that represents the queries to the AOL search engine in the ASPE system, the system is able to continue reading the stream (figure 4.5). The system is slower when garbage collection is applied due to the workload for the garbage collection system.

Because the initial state of all the operators is empty, lookup of reference data is initially very fast and the first mutations are accepted at a high rate. When more data is added into the state space of operators, reference lookups in operators slow down due to the number of tuples to search through and

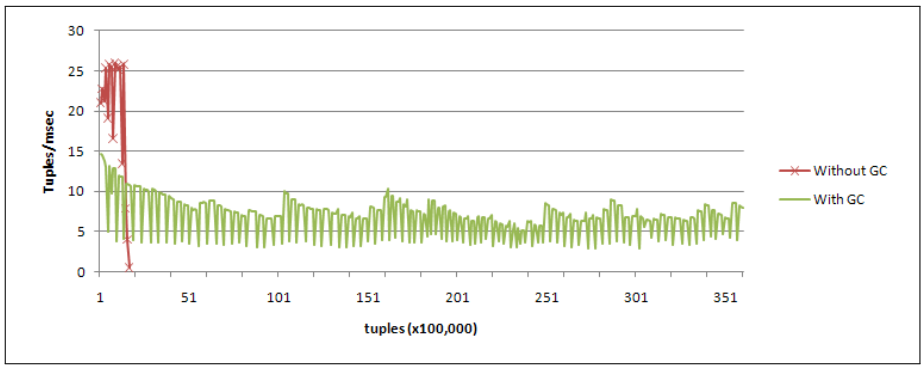

Figure 4.5: ASPE performance

mutations are accepted at a lower rate. The system eventually stabilizes the rate of mutations at an average of 5000 mutations per second.

Chapter 5

Conclusion

In this thesis, the idea of a Stream Processing Engine has been extended. The newly developed Analytic Stream Processing Engine (ASPE) accepts mutations to a virtual data storage instead of tuples. The ASPE uses the mathematical properties of statistical functions in order to aggregate data through continuous adaptations of the result. The tuples depicted by the mutations do no longer need to be stored.

Due to the absence of a central storage, the developed system is able to provide query answers without any locking of resources. Mutations to the virtual storage are used to incrementally adjust query result sets for a small number of statistical queries.

The ASPE uses operators which maintain their own storages of intermediate data called state spaces. To keep the state space of these operators in check a garbage collection scheme and a query optimization scheme are proposed.

The speed of the system in accepting mutations is primarily dependent on the number and the complexity of continuous queries in the system. Because adapting a query result set from a single mutation is a small amount of work, the system is able to accept a steady, high number of mutations per second.

A mutation to the virtual storage is directly propagated to query result sets. Therefore, the latency from accepting a mutation to the virtual storage and representing that mutation in the query result sets is very small.

The number of tuples in the virtual storage of the ASPE does no longer have an effect on response times of query result requests. A request for a query result always yields an instant response. No logic is needed to serve requests, since query results are adjusted when the virtual storage is mutated.

The ASPE is able to accept a massive number of mutations to a possibly endless number of tuples. The ASPE is able to extract near-real time statistics out of these mutations and serve these to a large number of users.

Chapter 6

Future Work

In this chapter, improvements to the ASPE are mentioned. The future work is divided into functionality improvements and issues which require research.

6.1 Functionality Improvements

The ASPE is a system which has been implemented as a proof of concept. For the ASPE to reach maturity, the system needs to be improved. The most important improvements are mentioned in this section.

6.1.1 Operators and State

Operators may maintain a state. The operator and its state are now intertwined; the Aggregate and Join operators themselves maintain the storage of tuples in their state. In order to enable different strategies to store the state spaces, the code for the storage of the operator state should be detached from the functionality of the operators. This enables different strategies of storing the state space to be accessed through a common interface.

6.1.2 Garbage Collector

The garbage collector scans all the tuples in the operator state periodically in order to find and delete immutable information. Currently, the garbage collector iterates over all tuples in the state space to find tuples that are outside of the mutation window. There should be a way to index the tuples in the state of the operator in order to scan for tuples outside the mutation window more efficiently.

6.1.3 Durability

Durability was out of scope for this research, however durability is important in databases and the ASPE is no exception. Durability can be achieved by keeping logs of all mutations for replay. Other strategies could serialize operator states to disk directly or a hybrid form of the above methods.

6.2 Research

Though the concept of the ASPE has been tested, the current ASPE is not optimal. There are some areas where a naive solution has been implemented in order to prove the concept the ASPE. In this section, unoptimized parts of the ASPE are mentioned, as well as possible solution.

6.2.1 Usability

The ASPE contains programmable operators. This gives the ASPE the ability to support many kinds of expressions. However, the operators are hard to program and debug. Writing a query for the ASPE requires a lot of debugging and knowledge of the system. This could be fixed by creating a declarative, SQL-like language with precoded operators.

6.2.2 Query Optimization

When a running continuous query is optimized, the optimization algorithm switches operator order in the query tree. As of now, two operators that require a state space (the Aggregate and Join operators) are not able to switch at runtime due to the inability to adapt the operator state to the new situation. Therefore, current optimizations are performed by keeping two parallel operator structures. New insertions are redirected to the optimized structure while the old structure is kept and maintained until all the data in the state spaces of its operators has become immutable. This approach is not able to produce results fast and could probably be improved by altering the state of an operator.

6.2.3 Ordered sets

The maintenance of ordered result sets is a hard problem [22]. The system does not provide the ability to efficiently sort data. Due to the high rate of mutations in the system, using b-trees to maintain an ordered set might not be a good solution due to the amount of effort needed to maintain b-trees.

The solution may be to order information into value intervals. By leaving the tuples within the intervals unordered, the amount of work at the arrival of a mutation is small. When streaming tuples to the user, the system needs to order only the tuples within the intervals, thereby reducing the amount of work at the arrival of mutations and limiting the amount of work at querying.

Bibliography

[1] CHUTE, C., MANFREDIZ, A., MINTON, S., REINSEL, D., SCHLICHTING, W., AND TONCHEVA., A. An updated forecast of worldwide information growth through 2011. Tech. rep., IDC/EMC, 2008.

[2] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107 – 117. Proceedings of the Seventh International World Wide Web Conference.

[3] ABADI, D., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the borealis stream processing engine. *CIDR* (2005).

[4] GOLAB, L., AND OZSU, M. Issues in data stream management. *SIGMOD 32*, 2 (June 2003).

[5] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M., HELLERSTEIN, J., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. Telegraphcq: Continuous dataflow processing for an uncertain world. *CIDR* (2003).

[6] ARASU, A., BABCOCK, B., BABU, S., CIESLEWICZ, J., DATAR, M., ITO, K., MOTWANI, R., SRIVASTAVA, U., AND WIDOM, J. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] LEWIS, P., BERNSTEIN, A., AND KIFER, M. *Databases and Transaction Processing*, international ed. Pearson Education international, 2001.

[8] SINGHAL, V., AND SMITH, A. Analysis of locking behavior in three real database systems. *VLDB* (1997), 40–52.

[9] POTIER, D., AND LEBLANC, P. Analysis of locking policies in database management systems. *ACM 23*, 10 (1980), 584–593.

[10] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *OSDI* (2004).

[11] H. YANG, A. D., AND R. HSIAO, D. S. P. Map-reduce-merge: simplified relational data processing on large clusters. *ACM* (2007), 1029–1040.

[12] RAFIQUE, M. M., ROSE, B., BUTT, A. R., AND NIKOLOPOULOS, D. S. Supporting mapreduce on large-scale asymmetric multi-core clusters. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 25–34.

[13] BABCOCK, B., DATAR, M., AND MOTWANI, R. Load shedding for aggregation queries over data streams. In *20th International Conference on Data Engineering (ICDE 2004)* (2004).

[14] REISS, F., REISS, F., HELLERSTEIN, J. M., AND HELLERSTEIN, J. M. Data triage: An adaptive architecture for load shedding in telegraphcq. In *In ICDE* (2004), pp. 155–156.

[15] TATBUL, N., AND ZDONIK, S. Dealing with overload in distributed stream processing systems. *VLDB* (2006).

[16] TATBUL, N., ÇETINTEMEL, U., AND ZDONIK, S. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 159–170.

[17] GOLAB, L., AND ÖZSU, M. T. Issues in data stream management. *SIGMOD Rec. 32*, 2 (2003), 5–14.

[18] COHEN, E., AND STRAUSS, M. J. Maintaining time-decaying stream aggregates. *Journal of Algorithms 59*, 1 (2006), 19 – 36.

[19] TATBUL, N., AHMAD2, Y., CETINTEMEL, U., HWANG, J., XING, Y., AND ZDONIK, S. Load management and high availability in the borealis distributed stream processing engine. Tech. Rep. Technical Report, ETH Zurich, 2006.

[20] AOL. Aol search data. `http://research.aol.com/pmwiki/pmwiki.php?n=Research.Research?action=downloadman&upname=500kusers.tgz`.

[21] AOL. Aol search data (copy). `http://www.gregsadetsky.com/aol-data/`.

[22] AGGARWAL, A., AND VITTER, JEFFREY, S. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (1988), 1116–1127.