

Flexible and scalable storage management for data-intensive stream processing

Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, Nesime Tatbul

Angaben zur Veröffentlichung / Publication details:

Botan, Irina, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. 2009. "Flexible and scalable storage management for data-intensive stream processing." In Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09, Saint Petersburg, Russia, March 24 - 26, 2009, edited by Martin Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, 934-45. New York, NY: ACM Press. <https://doi.org/10.1145/1516360.1516467>.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



Flexible and Scalable Storage Management for Data-intensive Stream Processing *

Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, Nesime Tatbul
Systems Group, Department of Computer Science, ETH Zurich
{irina.botan, alonso, peter.fischer, kossmann, tatbul}@inf.ethz.ch

ABSTRACT

Data Stream Management Systems (DSMS) operate under strict performance requirements. Key to meeting such requirements is to efficiently handle time-critical tasks such as managing internal states of continuous query operators, traffic on the queues between operators, as well as providing storage support for shared computation and archived data. In this paper, we introduce a general purpose storage management framework for DSMSs that performs these tasks based on a clean, loosely-coupled, and flexible system design that also facilitates performance optimization. An important contribution of the framework is that, in analogy to buffer management techniques in relational database systems, it uses information about the access patterns of streaming applications to tune and customize the performance of the storage manager. In the paper, we first analyze typical application requirements at different granularities in order to identify important tunable parameters and their corresponding values. Based on these parameters, we define a general-purpose storage management interface. Using the interface, a developer can use our SMS (Storage Manager for Streams) to generate a customized storage manager for streaming applications. We explore the performance and potential of SMS through a set of experiments using the Linear Road benchmark.

1. INTRODUCTION

Modern data stream processing applications involve multiple continuous queries that run in parallel, join live data with stored historical information, and are highly data-intensive, requiring temporary materialization of large windows as well as maintenance of large and highly dynamic operator state. Examples include highway traffic monitoring [3], Internet traffic analysis [9], log monitoring/mining [10], and scientific data processing [13]. These applications often operate under very strict performance requirements.

Although many DSMS solutions have been offered to date (e.g., [1, 2, 6]), none of them provide a clean and systematic approach

*This work was supported in part by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation under grant number 5005-67322.

to storage management. In these systems, the storage manager is tightly coupled with the query processing engine. Such a design not only makes the storage manager adhoc and inflexible, but it also severely limits exploiting the optimization opportunities of the applications to their full potential since a hard-coded implementation does not leave much room for optimization. In practice, being able to customize and tailor the storage manager to the requirements of the application is key to achieving good performance.

Interestingly, using a tunable storage manager separated from the query engine has been a fundamental design principle of traditional DBMS architectures and has been the basis for many useful performance optimizations (e.g., [8]). A first contribution of this paper is to argue that a similar design is needed for data streams. Hence, in the paper we propose such a separate storage manager and prove its advantages by showing how to make a storage manager tunable. By analyzing the patterns observable on data streams and the queries over these data streams, we have identified a set of important parameters that can be used to tune the performance of the storage manager. Based on this analysis, we have developed an advanced interface that can be used to generate highly efficient, customized storage managers.

Our approach of a tunable, customizable storage manager clearly borrows ideas successfully exploited for buffer management in traditional databases [8]. In this area, the paper makes additional contributions as the problem is quite different in the case of data streams. As in relational databases, the read patterns of streaming queries can be predicted in advance and can be used to select data structures, access paths, and indices. This approach makes even more sense in a streaming system, where continuous queries are known a priori and their more accurate static analysis is possible. Unlike relational databases though (where data is relatively more static and updates are less frequent), in a streaming system, the update patterns also play a key role, as they affect decisions on data layout to better support highly dynamic data movement. Considering update patterns in the optimization strategies is nontrivial and the paper discusses in great detail which parameters are relevant and how to use them to tune the storage manager.

The final two contributions of the paper are the system itself and its performance evaluation. The system we describe is called SMS (Storage Manager for Streams). SMS is a general-purpose storage manager for DSMSs that uses the storage parameters defined through our analysis and delivers a method for tuning them for performance. SMS is built on a well-defined and powerful interface so that SMS can easily and effectively be tailored to different application needs and can potentially serve as the underlying storage manager for any DSMS. To prove the advantages of SMS and the feasibility of the ideas explored in the paper, we have implemented the Linear Road benchmark on top of SMS. Our experiments show

that SMS can achieve a 2-6 factor of improvement over a “one-size-fits-all” baseline store implementation (i.e., not tuned well to application needs), measured using three different performance metrics.

This paper is structured as follows: In Section 2, some motivating application scenarios are described together with their storage requirements. In Section 3, we present our fine-grained analysis to identify the key storage requirements and their respective parameters. Section 4 focuses on the access pattern parameters and discusses a framework for the possible values these parameters can take. Section 5 presents our SMS architecture. In Section 6, we describe the implementation of the SMS store instances for different storage parameters, based on state-of-the-art techniques. Section 7 proves, through a benchmark study, that SMS can meet the storage requirements very well, and can significantly improve the query processing performance. Section 8 presents an overview of the related work. Finally, we conclude the paper with a discussion of avenues for future work in Section 9.

2. APPLICATION SCENARIOS

We are targeting data-intensive streaming applications, including but not restricted to applications that need to manage a large state during stream processing. Examples of such applications are:

Internet Traffic Analysis: Correlation of packet streams over wide-area networks requires keeping a significant amount of data available for joining over time/location/protocol [9].

Log Monitoring/Mining: Discovering new patterns or checking the presence of pre-defined patterns in data log streams involve managing large processing state [10].

Scientific Data Processing: Very high data rates of scientific experiments need to be processed under fairly tight time constraints in order not to overload the storage and network capabilities [13].

One challenging application scenario that we study in detail in this paper, is the **Linear Road Benchmark** [3], which was developed to evaluate the performance of a DSMS.

Linear Road simulates the traffic on a set of highways with segments, and provides variable tolling depending on traffic statistics and accident occurrences. Input stream consists of car position reports and queries. The system has to react to different patterns in the reports and answer the queries accordingly.

Linear Road’s design goal has been to stress all possible components of a DSMS, not just some operator or scheduling implementation. In particular, it provides a fairly comprehensive set of requirements to stream storage, some of which we briefly outline here:

- For computing the toll for a specific segment on a highway, the traffic statistics of that segment are used. Statistics are obtained from analyzing a window with traffic information (number of cars, speed etc) created for the past five minutes with a slide of one minute. Therefore, the traffic statistics information expire in the order that they were generated.

- Once an accident is detected, every vehicle that enters into a segment in the vicinity of that accident must be notified. This requires storing the accidents until they are cleared. Cleared accidents are determined by new values in the streaming data (i.e., cars that had previously stopped, started to move).

- Tolls assessed to a specific car need to be stored for the drivers to be able to later request their current balance of toll spendings. One solution for storing this information is to create (key, value) pairs, while using the new tolls to update the balance that has the appropriate key (the car id).

- The car position reports are used by multiple queries: accident detection, segment crossing detection and segment statistics computation. These three queries maintain windows of different sizes

on the same data, which for performance reasons could be shared.

- For the daily expenditure query, ten weeks worth of data needs to be stored (the tolls spent by all drivers in this time interval). Given its static nature and the fact that it is quite large, a solution is to use a relational database for storing it.

- As the information about the accidents in the past minute may not be available when a certain car crosses a segment, a synchronization condition on the store contents is required to keep the request waiting until the most recent accident data is available.

As discussed above, Linear Road presents a clear evidence that data-intensive stream processing applications may come with a wide variety of storage requirements. Providing a flexible and configurable storage management solution is key to meeting these requirements in the most effective and scalable way.

In the next sections, we will describe the three major building blocks of a decoupled storage manager for streams: (i) the language for specifying storage requirements (represented with a set of parameters) in Sections 3 and 4, (ii) the interface to communicate them in Section 5, and (iii) their internal implementations in Section 6.

3. MODELING STORAGE MANAGEMENT FOR STREAM PROCESSING

We first conduct a requirement analysis in order to identify the set of tunable parameters that a Stream Processing Engine (SPE)¹ can use to communicate its needs to a storage manager.

These parameters have threefold importance: (i) they provide the required data access functionality (i.e., basic per-item operations), (ii) they support the SPE in query processing and optimization (e.g., predicate push-down, push/pull models), and (iii) they offer support for reducing the storage management costs (e.g., lowering response times and memory consumption).

3.1 Architectural Parameters

A DSMS can be architected in two ways in terms of its processing model: push-based and pull-based. In the former way, input stream items are pushed through the query network (e.g., Aurora [1]), whereas the latter follows a more traditional way in which stream items are pulled from the source input stream, processed and stored in a result container to be further pulled by other operators (e.g., MXQuery [5]). A general-purpose storage system has to be able to support and combine both processing models under the same framework. Therefore, the first part of our analysis refers to how the data items get into the storage structures as well as from there back into the SPE, behavior captured by two parameters:

Role: This property shows how the data gets materialized from a stream into a store (Figure 1). A store can be either *active* or *passive*. An active store is directly connected to the source stream and pulls the items from it. In the case of a passive store, on the other hand, the SPE receives the items from the input stream and pushes them to be materialized in the store.

Access Model: This property shows how the data gets from a store to the SPE. There are two ways: *pull-model* and *push-model*. Pull-based stores wait for requests from operators, while push-based stores notify the engine about new available results.

3.2 Functional Parameters

In addition to architectural parameters, there are also certain properties that arise from a special functional requirement of an SPE:

¹We use the term SPE to refer to the query processing part of a DSMS.

Parameter	Description	Possible Values	Default Values
Role	How the data is materialized in a store	Active, Passive	Passive
Access Model	How the data gets from a store to the SPE	Push, Pull	Pull
Synchronization	If the store acts as a synchronization point	False, Synchronization condition	False (no wait)
Schema	Schema knowledge: number, names, type of attributes	No schema, Schema information	No schema
Persistence	Where the data is materialized (disk or memory)	Persistent, Transient	Transient
Read Pattern	Requests for reads from a store	Sequential, Random, Clustered	Random
Update Pattern	Requests for updates to a store	FIFO, RANDOM, IN-PLACE	RANDOM
Sharing	If the store is shared by multiple operators executing reads	Shared, Not shared	Not Shared

Table 1: SMS tunable parameters

Synchronization and Schema: SPEs usually have a *scheduler* whose job is to run operators in a certain order. One task that they cannot achieve though is to delay execution of an operator until a condition based on item values is met. An example extracted from the Linear Road benchmark is accident notification. A driver entering a new segment of a certain highway should be notified by accidents in the vicinity, an information which should be up to date (from the last minute); if not, the notification would be delayed until the condition is met. For implementing this requirement in Aurora, the authors had to add a new operator box, named Wait-For [4]. A materialized store with some synchronization capability would have made the creation of such an operator box unnecessary. That is, the SPE may specify a condition on the store which needs to be met before the consumer operator is allowed to retrieve data. Usually this condition is on some attribute values, which means that it is important to know the schema of the streaming items (if available). From this we extract two parameters: *Synchronization* whose value is represented by the synchronization condition and *Schema* representing the stored data’s schema knowledge: if all the stored items comply to the same schema or not, number of attributes, attributes names and types for each of the schemas. The schema information is also important for predicate push-down optimizations. Besides functionality, as it will be shown later in the paper, knowing that all items comply to the same schema helps the storage manager in making some implementation decisions.

Persistence: Queries involving archived data sometimes need items which are already stored in a relational database. Additionally, an SPE, based on the characteristics of the data (volume and/or how often it is updated, etc), could also specify that it should be stored on disk rather than in main memory. To account for these scenarios, we introduced a parameter named *Persistence*, with two possible values: *persistent* and *transient*.

3.3 Performance-related Parameters

Apart from providing the necessary architectural support and functionality, the storage manager should also provide performance. As stated before, we target data-intensive applications which have low response time and low memory consumption requirements. As we will show later in the paper, certain pieces of information help the storage manager to make decisions about how to help the SPE in meeting these requirements. The access patterns of the operators and sharing of materialized data are two of the most important parameters because, given their values, the storage manager will implement specific mechanisms for speeding up store operations.

Access Patterns (Read and Update): An operator’s storage requirements can be characterized by its *data access patterns*. By observing the read and update behavior of an operator on its internal state (or on intermediate results), we can depict some general patterns, which can be then used in implementations for performance. We will present a more detailed, fine-grained analysis of the read

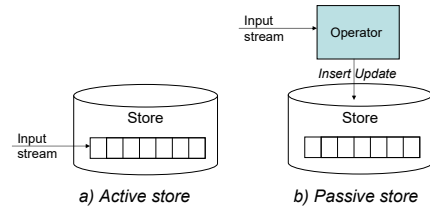


Figure 1: Role store property

and update patterns in Section 4.

Sharing: Continuous queries show high similarities and this observation was used by the SPEs for multiquery optimization. The performance of running multiple continuous queries in parallel could be improved by sharing computation, and therefore, intermediate results and operator states. As an example, we consider computing the accidents in Linear Road. As outlined in Section 2, there are two queries which share the accidents information: the one which notifies other cars about the accidents, and another one which computes the tolls assigned to a segment based on accident vicinity. The *Sharing* parameter can be used to support an SPE in performing its sharing-based query optimizations.

Table 1 provides a summary of the SMS parameters that we identified in this section. In the next section, we will further elaborate on access patterns and sharing, and show how they can be set to capture performance-related behaviors. We only concentrate on the performance-related parameters as they are tunable and have a high impact on performance. The architectural and functional parameters have generally binary values and the decisions for their implementations are straightforward.

4. ACCESS PATTERNS

In a DSMS, there is a continuous flow of data items that enter the system, get processed through the queries to further produce results that leave the system. During this process, the data items may get materialized (either permanently or temporarily) in different types of stores. Additionally, in case of temporary materialization, after a while, the data items become outdated (i.e., no longer needed by the queries) and therefore must be deleted from the stores. An SPE needs to access each of these stores in different ways during query processing, determined by the access patterns of the query operators. In this section, we present the access pattern parameters together with the possible values that they can take.

We consider streams as possibly infinite sequences of data items. Each item has a set of attributes, which are the same for all items in a stream if they comply to the same schema. When materialized into a store, each data item is assigned a unique id, which is further used to identify the item in the store as are all its attributes.

SELECT Attr1, Attr2 FROM Stream	SELECT Attr1, Attr2 FROM Stream WINDOW 24 HOURS	ON Stream S DELETE FROM Store W WHERE W.Attr1 = S.Attr1	ON Stream S UPDATE Store W SET W.Attr2 = W.Attr2 + S.Attr2 WHERE W.Attr1 = S.Attr1	SELECT * FROM Store
a. Never Expire	b. Ordered Expire	c. Unordered Expire	d. Replaced Expire	e. Never Consume
SELECT Attr1, Attr2 FROM Store WINDOW 24 HOURS	SELECT FIRST(S.Attr1), LAST(S.Attr1) FROM Store S WINDOW 24 HOURS	SELECT Attr1, Attr2 FROM Store WINDOW 24 HOURS	SELECT Attr1, AVG(Attr2) FROM Store WINDOW 24 HOURS GROUP BY Attr1	SELECT S.Attr1 FROM Store S WHERE PREV(S.Attr1) != S.Attr1 AND S.Attr1 < NEXT(S.Attr1)
f. Ordered Consume	g. Eager Consume	h. Sequential Read	i. Random Read	j. Clustered Read

Table 2: Query examples for each access pattern

Data items in a store can be accessed in two alternative ways: value-based and id-based. In the value-based access, given a set of attribute values, the matching items are returned; in the id-based access, the item with the corresponding id is returned.

Furthermore, there are two major categories of access patterns: read patterns and update patterns. Read patterns refer to the way data items get retrieved by the operators, while update patterns refer to how operator states and inter-operator stores get modified as a result of new data arrival, query processing, as well as outdated data eviction.

4.1 Update Patterns

Since data stream items arrive and need to be processed in a pre-defined order (either order of arrival, or given by the SPE), new data items' insertion into a store is always done in an append-only manner. Therefore, in this section, we will only focus on the update patterns which occur in the form of deletes and value replacements.

A store gets updated with deletes or value replacements in two occasions: (i) as a result of data expiration, and (ii) as a result of data consumption.

Our analysis of data expiration is similar to that of Golab and Özsü [12]. The fundamental difference is that we are using the results of our analysis to determine how to configure the stores, while the related work uses them for operator implementation as part of an SPE.

To illustrate, consider two operators (Producer and Consumer) connected as shown in Figure 2. The Store between them temporarily keeps the intermediate query results that are generated by the Producer to be later used by the Consumer. The Producer, in addition to the regular append-only insertion of new data items, has an update pattern on the Store denoted by UPD1, while the Consumer has an update pattern on the Store denoted by UPD2 and a read pattern denoted by READ. UPD1 is a result of data expiration, while UPD2 is a result of data consumption.

Next, we will present the possible update operations that the Producer (UPD1) and the Consumer (UPD2) operators may impose on the store and their semantics (for each of the patterns we provide a simple query example written in an SQL-like language in Table 2). The combinations of these patterns will become the update patterns of the Store itself.



Figure 2: Operations on a store

4.1.1 Producer Operator Update Patterns

Data items in a store may expire in four different ways:

Never Expire. The Producer generates a stream of results which are materialized in Store, but never expire. This means that there is no UPD1. For example, if it performs a selection operation (Table 2.a.), the selected items are inserted in the Store as regular append-only, and they never expire. In theory, the resulting data items could be kept in the Store forever. In practice, the Consumer operator is to determine when some items are no longer needed and can be deleted (will be described in the next subsection).

Ordered Expire. In this case, the data items in the Store become outdated, and therefore deleted, in the order that they were created. For example, if the Producer operator selects some attribute values from a moving window of 24 hours, the query results materialized in the store will expire as the window moves forward (Table 2.b.).

Unordered Expire. In some cases, the update pattern may depend on the values of the newly arriving input rather than its arrival order. In other words, the outdated items are determined by the values of some attributes. Since insertions are always append-only, the items in the store will not necessarily be physically clustered by their key values. Therefore, value-based delete requests will usually be translated into randomly ordered deletes (i.e., two successive delete requests will not follow a specific order). An example is provided by the statement in Table 2.c., which deletes from the Store the items having the same values for attribute "Attr1" as the items in the input stream.

Replaced Expire. Some store items expire when a new item replaces their attribute values. (e.g., if the Producer executes the query in Table 2.d., it will update the value of attribute "Attr2" in Store with new data from the input stream).

Note that Unordered Expire and Replaced Expire patterns require value-based access to the data items in the Store.

4.1.2 Consumer Operator Update Patterns

The updates are generated not only as data items expire, but also as they get consumed. Some of the consumer operators follow the usual stream-based semantics, i.e., they "look" at the input only once. In this case, some data items may redundantly remain in the system, resulting in waste of memory space. We solve this problem by having the consumer operators mark the processed items as "consumed". Unlike in data expiration, these items need not be eagerly deleted; but instead, they could be removed from the stores lazily. Lazy removal is achieved by periodically running a "garbage collection" process to remove the marked items from the store.

Data items in a store, which have already been processed by all of its consumer operators, can be marked for eviction in three different ways:

Never Consume. In some cases, the Consumer operator may not be able to decide if any of its consumed items will be needed in the future. If this is the case, then no item is marked as consumed. An

	Never Consume	Ordered Consume	Eager Consume
Never Expire	NO-UPDATE	FIFO	FIFO/RANDOM
Ordered Expire	FIFO	FIFO	FIFO/RANDOM
Unordered Expire	RANDOM	RANDOM	RANDOM
Replaced Expire	IN-PLACE	IN-PLACE	IN-PLACE

Table 3: Store update patterns

example would be when the Consumer operator executes a repeated scan of the store (Table 2.e.).

Ordered Consume. When a sliding window operator moves the window, the outdated items are marked as consumed (if for example the Consumer operator executes the query in Table 2.f.). This is done in a FIFO manner, since older items are consumed earlier than the newer ones.

Eager Consume. In some cases, a windowed Consumer operator can take an eager approach in marking the consumed items in the window. For example, when the window (materialized in the store) grows to large sizes, the operator may determine that some of the items inside the window will no longer be needed and therefore, can be immediately marked as consumed. This approach not only reduces memory consumption, but also decreases query processing time, since window lookups for smaller windows take less time. An example is provided in Table 2.g., in which the query only requires the first and the last items in a window (all items in between could be marked for removal).

4.1.3 Store Update Patterns

The update patterns of the Producer and Consumer operators can be combined to assign a single update pattern to a given Store. For our example in Figure 2, the patterns for UPD1 and UPD2 would determine the update pattern of Store. In this section, we consider all combinations of update patterns due to expiry and due to consumption, and come up with a number of possible update patterns for stores. Table 3 presents these combinations.

In general, the update pattern of the Store is determined by the update pattern of the Producer operator. The reason is that expired data items must be immediately removed from the Store to make sure that the Consumer operator has accurate inputs. On the other hand, if the Producer has the Never Expire pattern, then the update pattern of the Store is determined by the update pattern of the Consumer operator.

Except for Replaced Expire, all other Producer updates are in the form of delete operations. Therefore, for the value replacement case, we need a special store update pattern, which we call IN-PLACE. For the other pattern combinations, we always choose as the store pattern, the operator update pattern with the most constraints. For example, a combination of Unordered Expire producer update pattern and Ordered Consume consumer update pattern will result in a RANDOM store update pattern, since the former imposes the support for deletes in random order.

From the producer-consumer update pattern combinations, we identify four types of store update patterns:

- **NO-UPDATE.** This store update pattern has no fixed behavior imposed by the Producer-Consumer operators. This is a special case for which a couple of different approaches can be taken, like: to archive materialized data (on disk) or to consider a FIFO pattern with some lifetime specification.
- **IN-PLACE.** This store update pattern is for stores that allow value replacements by key. If the key of an updated item does not already exist in the store, then that item is inserted.
- **RANDOM.** This store update pattern is called RANDOM to

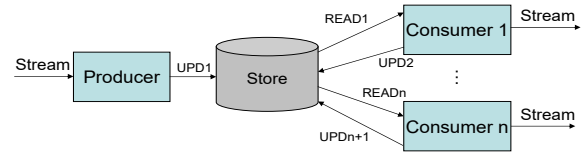


Figure 3: Shared store

account for the unordered execution of the delete operations.

- **FIFO.** The most common combination of operator update patterns observed in streaming applications is represented by a mix of inserts at the end and deletions from the beginning. For this specific behavior, we define FIFO, a store update pattern which favors queue-like operations. In this type of stores, either items never expire, or they expire in an ordered fashion. For Eager Consume, we can take two approaches: FIFO, when memory consumption is low and we can ignore items being marked as outdated, or RANDOM, to support eager deletion of items.

The implementation of the store update patterns listed above will be described in Section 6.

4.2 Read Patterns

As in related work [8], we do an analysis of the read patterns, in our case, in a streaming environment.

As shown in Figure 2, the way the Consumer operator requests items from the store determines the read pattern. Although windows are usually stored in main memory, executing a repeated scan of the complete window to find the requested items is not an efficient solution. Depending on the Consumer operator’s read access patterns, certain optimizations can be applied. In this subsection, we identify these patterns and in Section 6 we discuss their optimized implementation.

Streaming operators read data items from a store in three different ways:

Sequential. Some Consumer operators access the data items in a store sequentially. Selection inside a window is an example for sequential read. In this case, the items in the specified window will be accessed one after another (e.g., the query in Table 2.h.).

Random. Some operators access the data items in a store in a random order. For example, an Consumer operator executing a windowed group-by aggregate operation will access the items of a window in random order of the groups (Table 2.i.).

Clustered. In a clustered read access, there is locality of requests grouped around some items. For example, the query in Table 2.j. executes a series of read operations for previous, current and next item given the current position of a sequential cursor.

4.3 Sharing and Access Patterns

In the previous subsections, we made an enumeration of different read and update patterns. We were considering the simple case in which there was only one consumer operator. For optimization purposes though, stores can be shared among multiple consumers which require access to the same input data (Figure 3).

For determining the update pattern of a shared store, we propose a simple and safe (i.e., that meets all the constraints and produces correct answers) algorithm: we first determine the combined update pattern of the consumer operators, and then use Table 3 to obtain the final update pattern representing that store.

The three update patterns on the consumer side (i.e., Never Consume, Ordered Consume, or Eager Consume) yield the following possible combinations:

* If at least one of the consumer operators has a Never Consume update pattern, then the combined update pattern should be Never Consume.

* If all consumers follow an Eager Consumer update pattern, then the combined update pattern may also be Eager Consume for performance reasons (less memory consumption). On the other hand, a less constrained pattern (e.g., FIFO) could be used when there is enough memory.

* All other cases generate Ordered Consume update pattern.

Sharing has an important impact on performance. This is because different operators need different subsets of the input, and therefore, there may be many items in the store that are not needed by a certain operator. In this case, knowing how each operator reads the data is important for speeding up the search. Therefore, the read pattern of a store is determined by the read patterns of each of its consumer operators. In other words, the store must provide the storage structures (e.g., indices) to support the read patterns of all of its consumers.

5. SMS ARCHITECTURE

As outlined in the introduction, our work does not stop at identifying the parameters needed to tune storage in a DSMS, but it takes the next logical step to establish an architecture for decoupling the storage concerns from the processing concerns in order to allow for better optimization, generality and extensibility.

Figure 4 shows how a DSMS can be split in two components: the stream processing engine (SPE) and the stream storage, which we call SMS. The SPE is mainly in charge of query processing and optimizations, and uses SMS to perform all of its storage-related tasks (i.e., storing and retrieving data items as needed by the queries). As such, SPE and SMS together act as a complete DSMS.

Given an application, it is the SPE's responsibility to analyze its requirements and then implement it (usually through a query network). During this process, the SPE uses our analysis of the parameters to determine the access patterns of the operators and to express its own architectural and functional requirements using the parameter values provided by SMS. The resulting combinations of values are then communicated through the dedicated interface to SMS, which, in turn will create store instances that meet the SPE's requirements.

In this section, we will present how SPE and SMS interact and how the detailed architecture of SMS looks like.

SMS consists of four main components:

Store Instance

A Store Instance represents a data source. It is characterized by the way its data is inserted, organized and updated. A store instance is configured to provide the required functionality through the set of

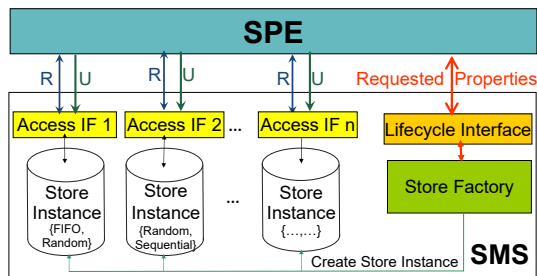


Figure 4: DSMS architecture with SMS

parameters specified at its creation time (some examples are shown in parenthesis in Figure 4). Furthermore, the implementation of a store instance ensures that both the response time and the memory footprint are low. We would like to note that a store instance could either reside in memory or on disk, which is a property that is specified through the *Persistence* parameter at creation time.

Lifecycle Interface

The Lifecycle Interface is used by the SPE to create and gain access to store instances. Through this interface, the SPE communicates to SMS the properties of the store instances that it would like to create through a set of parameters such as the access patterns. These parameters have been identified as a result of the detailed analysis conducted in the previous sections.

This interface exports a set of operations that are used for the management of store instances:

createStore(Properties) returns a *StoreInstance* that represents the implementation of the required properties. The new store instance is added to the list of currently available stores.

removeStore(StoreInstance) Destroys a store instance as requested by the SPE.

registerIndex(IndexSchema, StoreInstance) This method is used for specifying an index schema, that is used for creating indexes, to allow value-based replacements and reads (the IndexSchema will be described later in the paper).

Store Factory

As its name implies, the Store Factory is the component of the architecture that is in charge of creating, managing, and deleting store instances. The decisions regarding the implementation of the store instances are made by the Store Factory, which receives the requested parameter values through the lifecycle interface and uses rules for creating different store instances given the combinations of the parameters' values. (e.g., if a FIFO update pattern is specified in the properties, then the Store Factory applies the respective implementation for this type of update pattern).

Access Interface

The Access Interface enables the SPE to access a specific store instance. It provides basic per-item (and per-attribute) operations such as read and update. The supported operations depend on the actual parameter settings done at the instance creation time.

In the following we will present the interfaces exported when giving values to the store parameters. The *Sharing* and the *Synchronization* condition do not export any interface, as they are used for organizing the internals of the implementation.

Insertion.

Interface PassiveRole { bufferItem(Item) }

The *bufferItem* method is exported if the *Role* parameter is set to *Passive* value. Otherwise, this method is used internally by the store instance.

Update. In Table 4 we present the interfaces for each of the three store update patterns (NO-UPDATE obviously does not export any interface explicitly).

In practice, we don't expose an explicit consumption interface to the consumer operator, since the store update pattern and the read operations determine consumption.

The *update* method is exported if IN-PLACE value for the update pattern is received. *Values* are applied on the index determined by the *IndexSchema* to search for the item to be replaced with *NewItem*.

Read.

The *retrieve* method is exported if the requirements specify that

	FIFO	RANDOM	IN-PLACE
Exported methods	deleteUpTo	deleteItems	update
Parameters	ItemId	ItemIdsList	IndexSchema, NewItem, Values

Table 4: Update interface

	IndexRead	RandomRead	SequentialRead
Exported methods	retrieve	getItem getAttribute	getNextItem getNextAttr
Parameters	IndexSchema, Values	ItemId AttributeId	-
Return values	ResultSet	Item Attribute	Item Attribute

Table 5: Read interface

further requests will use an index. *Values* are applied on the index determined by the *IndexSchema* to search for the items returned as a result. The *ResultSet* can be further iterated to extract individual items.

Out SMS architecture provides a clean separation between the processing and the storage aspects of a DSMS. The ability to instantiate multiple, differently tuned store instances allows the fine-grained adaptation to the storage needs of an application, while enabling global storage optimizations via the store factory.

6. STORE INSTANCE CREATION AND IMPLEMENTATION

We have presented so far the "language" that the SPE can use for communicating with SMS and the interface it can use to do so. The next logical step is to follow the decision process that SMS executes when interpreting the received information. Given the parameter values, the storage manager configures a store instance that meets the requirements, while trying to provide an optimized implementation.

In the following we will present the decisions and actions that SMS takes when faced with each of the parameter values. Due to space constraints, some implementation details are not included.

6.1 Architectural Parameters

Role and Access Model: A passive value for the role, determines the store instance to export the "bufferItem" method to be used by the SPE to push items. When the value is set to active, the SMS will call a predefined API provided by the SPE to retrieve data items from the input stream. This interface should also provide a push operation to an output stream when the access model of the store is set to *push*. A *pull* value for the access model determines the store manager to export the read operations determined by the registered read pattern(s). The default values for the role and access model parameters are *passive* and *pull* respectively.

6.2 Functional Parameters

Persistence: When receiving value *Persistent* in a createStore call, the storage manager expects some information regarding the RDBMS: server URL, port, table name etc. It therefore creates a special kind of store whose job is to redirect all the requests from the SPE to the specified database. One challenge in the implementation is to translate the requests into SQL queries. Our current solution offers a simple mapping from (attribute name,value) pairs

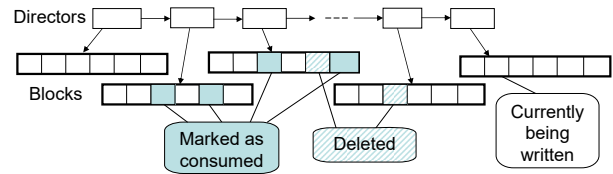


Figure 5: RANDOM store update pattern implementation

to SELECT statements as well as key-based updates. The storage manager has the flexibility to decide on when to persistently store the items, if it should keep them locally using caching techniques. That way, it relieves the SPE from the burden of managing these tasks itself (in the future we plan to further investigate the challenge of correctly and efficiently supporting join operations between persistent and live data) The value *transient* tells the storage manager to create its own main-memory based implementation of the store. The default value for the persistence parameter is *transient*.

Synchronization: If a synchronization condition is present, a special mechanism based on blocking/notification is employed for delaying reads until the condition is met. For example, a synchronization condition may look like: ("minute", eq, <minute_value>), meaning that the store should notify whenever there is data for the given minute (<minute_value>).

Schema: The next parameter evaluated is the schema(s) information (if present). The metadata about the attributes and types is stored. One important piece of information is if all the items in the stream comply to the same schema. If no schema information is available, the default action we take is to consider that the contained items may have different schemas and no further assumption is made.

6.3 Performance-related Parameters

In implementing the internals of a store, we used state-of-the-art techniques to speed up access operations as well as to keep low memory consumption. The memory usage vs response time trade-off applies here as well and we do have in mind dynamic decisions based on runtime statistics, but for the purpose of this paper we only focus on a best-effort static decision based solely on query analysis.

6.3.1 Implementation of the Update Patterns

The four basic write operations involved in any update pattern are: insertion, deletion, value replacement and garbage collection. We call *garbage collection* the process of physically removing the consumed and/or expired items from the store's internal structures. Each of the update patterns requires only a subset of the first three, while garbage collection is a required operation for avoiding unnecessary memory usage.

RANDOM Store implementation

Our RANDOM store implementation is depicted in Figure 5. This update pattern has the most constraints as it requires delete operations anywhere on the live (items that are neither expired nor yet consumed) part of the stored data as well as insertions and garbage collection.

The most suitable data structure for a RANDOM store is a linked list, because it is very flexible and easy to manage in the face of high-speed updates (i.e., insert and delete operations). A linked list allows the size of the materialized data to dynamically grow and shrink as needed. Furthermore, processing updates one item at a time is not a good option when dealing with fast updates (creating a

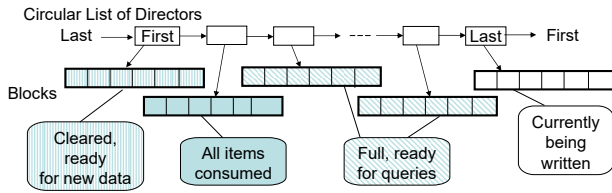


Figure 6: FIFO store update pattern implementation

new entry for each arrival dramatically decreases performance). In order to support high insert rates, we use a linked list of *directors* to *blocks* of items, in which items are ordered by their insertion times². A block itself is implemented as an array of items. Not only is the array implementation simple, but also provides fast random access. Each block has a fixed size, representing the number of items that it can accommodate. When a new item is to be inserted, it is redirected to the most recently created block in the first free position. If all blocks are full, a new one is created and the process continues.

When an item is to be marked as expired, its corresponding block is located using the list of directors and inside the block the item is marked as deleted. Any subsequent request for an expired item will fail. For fast access to a certain item we use a combination of logical (used by the SPE in requests) and physical Ids (used internally) with simple mappings based on arithmetical operations for reducing overhead.

An item or a set of items, can be marked as consumed in the same manner as expired items. Expired and consumed items are physically deleted from the store lazily, when the system becomes short of memory to make space for the newly arriving items. There are two approaches to garbage-collect the marked items: (i) only blocks which are fully consumed (i.e., all items in the block are marked) are removed from the linked list by removing the respective directors; or (ii) all the blocks are shrunk to physically remove all the consumed items in the system. The second approach is more aggressive as it also garbage-collects the partially-consumed blocks. Outdated items in a shared store are determined as the intersection of the item ids marked as consumed by all the consumer operators.

FIFO Store Implementation

One of the shortcomings of the RANDOM implementation is that in its context, blocks are continuously deleted and created. Furthermore, a block has to be locked for access, because a read operation may be requested for a block which is concurrently being written. This has a very important impact on performance with the continuous flow of items.

A consequence of having a FIFO update pattern is that consumed blocks may be reused instead of being deleted. That is, a block which has all its items marked as consumed, can be reused by replacing the consumed items with the new-coming ones, therefore avoiding deleting it and probably creating a new one later on. This is achieved by creating a circular linked list of blocks (Figure 6).

Furthermore, a read can only be executed after the requested item has been materialized (“read-behind”). In this case, the synchronization is achieved based on the *id of the last materialized item*. Therefore, we can make the block lock-free.

Specifying which items are consumed is a very simple task: the

²This is somewhat similar to the “sub-window” implementation in related work [11].

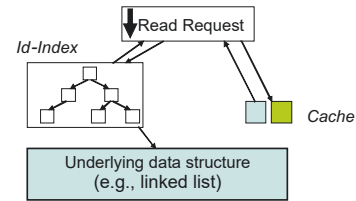


Figure 7: Speed-up read operations

smallest id of the last reported consumed items (in the sharing case) is computed, and all items up to that id are considered ready for garbage collection.

IN-PLACE Store Implementation

In the case of the IN-PLACE update pattern, we have no deletions, sometimes inserts and mainly replacements and reads. If we would use a single block for all the data items, reads would be executed really fast, but the block would have to be locked as replacements can occur in the meantime. Therefore we use a set of blocks, to reduce congestion (this resembles to horizontal partitioning in relational databases). This update pattern supposes the existence of a value-based index for locating the items that need to be updated. The value-based index is presented in the following subsection.

6.3.2 Implementation of the Read Patterns

To avoid a rescan with each new request for an item, the access pattern can be used to build specific indexing and caching structures on top of the layout.

Value-based Indices

As in the case of value replacement, value-based access requests the existence of a special structure which we call a *value-based index*. The schema information becomes very important for the implementation.

A value-based index is created using the predicate push-down feature. That is, the SPE pushes the *where* clause of a query operator to the storage level. First of all, the predicate is translated into an *index schema*, which the SPE registers on the store instance. This means that the engine specifies a set of attributes (i.e., columns) and logical operator pairs.

For example, consider the following query: *What is the previous minute’s “min” toll for the segment “seg” in direction “dir”?*

The conjunction-based predicate is translated into an index schema with the following structure: $((min, eq), (seg, eq), (dir, eq))$. The store will then create an index structure based on the three attributes. When a request identified by the index schema is sent, the values are applied to the index using the logical operators and the items are returned.

Read Patterns

Redirecting the item requests to the appropriate blocks may lead to overhead. Therefore, as shown in Figure 7, a store implementation uses a cache which holds the last block which was involved in a read operation. When an item is requested from the store, this cache is probed if it contains the corresponding block. If so, the cached item is returned. On the other hand, if there is a “cache miss”, then we need to locate the block which contains the requested item from the linked list. Because we want to avoid a complete scan of the list of blocks, we use a special structure which we call *id_index*. This index is a tree-based implementation of a map that is optimized

for locating the block an item belongs to. Consequently, the corresponding block replaces the current one in the cache. Sequential reads are facilitated by the presence of the cache, in which case the same block can be used for all subsequent requests going to that specific block. Having the actual information though determines that the id-index is removed, as it is not needed anymore: when an item is not in the current processed block, the linked list navigation methods can be used to move to the next block. Removing the index reduces the number of operations executed for a read, and avoids one synchronization point.

In the case that there are more readers subscribed to a store, each of them will have its own last accessed block saved in the cache.

Random access is facilitated by the id-index. In this case, we can skip probing the cache, as any block could be accessed next.

Having information about the schema of the input items is of great help when the SPE requests for the value of a certain attribute. This is achieved by allowing an identification scheme that locates attributes inside items. Our previous description about locating items does not work in this case. For this reason we create another structure for directly locating attributes. This implementation only works if all the items comply to the same schema.

The clustered read access is a pseudorandom access with some locality. Therefore, of great help will be the id-index backed up by the caching technique because two subsequent requests have a high chance of reaching the same block.

7. PERFORMANCE STUDY

In this section, we will show the performance of our approach through an experimental study on the Linear Road stream data management benchmark [3]. The benchmark is run on MXQuery [5], a Java-based open-source XQuery engine extended with window functions for stream processing. MXQuery uses SMS as its underlying storage manager for all of its storage-related tasks.

There are two main goals of this performance study:

- We would like to show that our fine-grained parameter analysis meets a broad range of requirements, while value tuning improves the overall query processing performance in terms of response time and memory consumption.
- We would like to analyze the sensitivity of the query processing performance to changes in some of the storage parameters.

7.1 Performance with Linear Road Benchmark

7.1.1 Linear Road Benchmark

In Linear Road, the input stream is composed of car position reports (each car reports its position every 30 seconds) and queries. An engine that implements the benchmark has to distinguish between the different types of streaming items and execute the queries accordingly.

The benchmark involves all in all, five queries: Accident Notification - drivers are notified of accidents in their vicinity when crossing to another segment on the highway; Toll Notification - drivers are notified by the toll of the segment they are entering; Balance Query - a driver can request its current balance of assessed tolls; Daily Expenditure Query - how much has a driver spent on a particular day in the past 10 weeks (Travel Time Estimation query was not included in any published implementation so far).

The measure of this benchmark is given by the *load level*, representing the number of highways that can be handled by the query processing engine. A certain load level is considered to be achieved, when all the queries involved in the benchmark are answered within at most 5 seconds after the request entered the system.

7.1.2 Experimental Setup

Our Linear Road implementation contains a set of queries connected by SMS stores as shown in Figure 8. Except for "Historical Tolls", all other stores reside in main memory.

There are three major types of SMS stores involved in our implementation: a persistent historical store for the tolls assessed to cars in the previous 10 weeks; two stores, which because of their nature, are implemented as main memory relations (allowing key-based updates and reads); and a set of "streaming" stores.

For the "Historical Tolls" we have created a Store Instance which has value *Persistent* for its *Persistence* parameter. Through our design, we have offered therefore a means for storing the data on disk, in this particular case using a MySQL database instance.

The two main memory relations (noted with I and II in Figure 8), both store items that are updated and/or retrieved by key (vehicle id) Given these requirements, the most suitable implementation is offered by the one corresponding to the IN-PLACE update pattern. These two stores have value "passive" for the *Role* parameter.

All the "streaming" stores are implemented as "active" stores (they pull items from their input streams of items):

Input Store holds items from the input stream sent by the Linear Road's data generator. It is shared by three queries (i.e., Car Positions query, Balance query, Daily Expenditure query).

Car Positions Store holds the car position reports used for computing the accidents and the per-minute statistics, as well as to determine which cars are crossing to a new segment. Therefore, three overlapping windows are open on top of this store. The largest window is the one for computing the statistics per segment (5 minutes).

Car Positions to Respond Store contains the position reports of the cars which changed their segment so that the drivers can be notified about relevant events (accidents or tolls charged).

Accidents Store keeps the data items that describe the accidents. It is shared by the accident notification query and the query that calculates the tolls.

Segment Tolls Store contains the per-minute, per-segment tolls. The "Accidents" and "Segment Tolls" stores are examples of using *synchronization* as both contain a condition on time (a request in a certain minute is blocked until data is available for that minute).

The streaming stores are the ones we focus on in this performance study, as this type of store is typical in general stream processing settings, and moreover, they do not have specific requirements, leaving room for different implementations and optimizations.

The experiments were run on a Linux machine with a 2.2GHz AMD Opteron processor and 4GB of main memory. SMS has been implemented in Java. For this study, we used Sun JVM Version 1.5.0_09 (we kept the default settings of Java garbage collection). The maximum heap size that represents the available memory was set to 3GB.

One experiment runs for 3 hours, until all items from the input are processed, unless the maximum amount of memory allowed is exceeded. The input load (number of streaming items per second entering the system) increases during the execution. In order to be able to demonstrate the full impact of our algorithms, in the experiments, we set the benchmark load level to 2.5 (sufficient to stress the system such that it had to fully utilize the allocated system resources).

In our first set of experiments, we used several metrics as performance measures, including: **the average and maximum query response time, the number of alerts which exceeded the 5 seconds threshold, and the total amount of memory consumed.**

The query response times are measured for the toll alerts, as they represent the majority of the alerts in the system and are generated

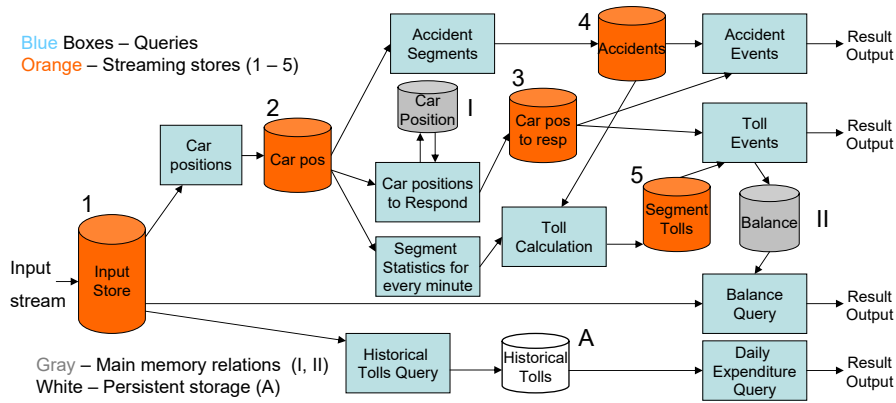


Figure 8: Linear Road implementation

on the query path with the highest computational load.

Our first attempt was to implement a simple queue as the underlying structure. The continuous items' deletes and inserts, as well as numerous read operations (MXQuery is a pull-based engine) made the store very inefficient as no special optimizations were conducted to support them. Of course, this implementation failed. Therefore, we used as the *Baseline*, an optimized version of a general streaming store (Figure 9). It is based on a RANDOM update pattern implementation (the most general, no assumptions), containing both an id-index for random access as well as caching techniques for speeding up reads (again, no assumption about the read patterns).

In the experiments we compare three different configurations:

- **Baseline** In this configuration all the stores in the query network are implemented as *Baseline*
- **Baseline + Update pattern.** In this configuration, we use the information about the update patterns and change the implementation of each store to reflect its actual update pattern (read pattern stays Random for all).
- **Baseline + Update pattern + Read pattern.** This configuration adds the read pattern knowledge, therefore determining some of the storages to change implementation to either Sequential or Clustered.

7.1.3 Performance

In this subsection we examine the performance of SMS when access patterns are tuned to the query requirements. First we consider the Baseline and run the benchmark in this configuration. Next we introduce the update pattern and present the new configuration. Fi-

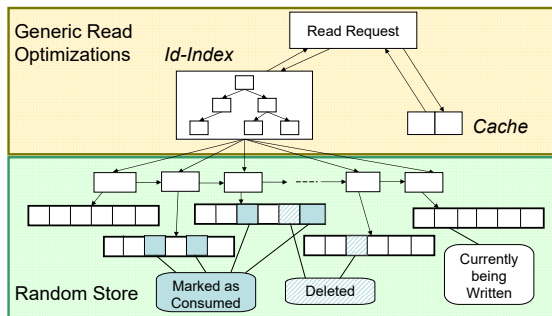


Figure 9: Baseline implementation

Results Configuration	Average response time (seconds)	Maximum response time (seconds)	Alerts over 5 seconds	Percentage of total alerts
Baseline	3.17	57	767,182	13.4
Optimized for update pattern	0.715	29	190,753	3.3
Optimized for update and read patterns	0.52	28	126,688	2.2

Table 6: Average and maximum response times

nally we add read pattern knowledge.

Analyzing the queries, we determine that the streaming stores 1, 3, and 5 (i.e., Input Store, Car Positions Store, Car Positions to Respond Store, and Segment Tolls Store) have a FIFO update pattern, whereas store 4 (i.e., Accidents Store) has a RANDOM update pattern (see Figure 8) since accidents are not necessarily cleared in the order that they were detected. As the Baseline considers RANDOM update pattern, we keep this implementation for the Accidents Store. Knowing that the access pattern is FIFO helps reducing the congestion, as individual blocks need not be synchronized anymore.

Next we tune our stores to exploit the read pattern knowledge. For this, stores 1 and 3 are configured as FIFO update and Sequential read; store 2 is configured as FIFO update and is shared, therefore offering three access patterns for each of the consuming operators: Random (for the "Segment Statistics for every minute" box), Sequential (for the "Car Positions to Respond" box) and Clustered (for the "Accident Segments" box); store 5 is configured as FIFO update and Random read; and store 4 is configured as RANDOM update and Random read.

Table 6 shows the performance results of our SMS approach compared to the Baseline (we present the median results of multiple runs for the experiments). As expected, the Baseline configuration behaves very poorly, because over 13% of the alerts come too late. The optimizations made possible by our parameter tuning reduced the average response time by a factor of 6 (factor of 2 for maximum response time). Furthermore, the number of toll alerts that exceed the 5 seconds threshold is reduced from 13% to 2%.

For the three configurations, we computed the average memory consumption: every minute, we checked the amount of memory used, sum that up and divide it by the number of minutes (180 for three hours). The average memory consumption is approxi-

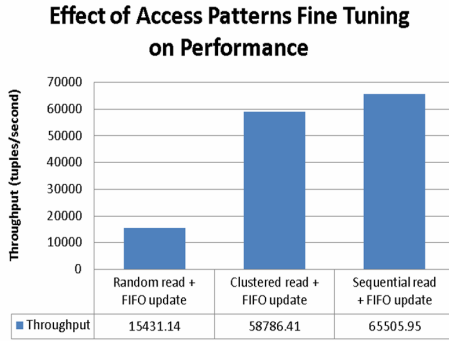


Figure 10: Effect of read patterns on performance

mately the same for all the configurations. The maximum memory used for (i) Baseline was 1.3GB, for (ii) Baseline with update pattern knowledge was 1.2GB and for (iii) Baseline with read and update knowledge, was 1.4GB. Because the Baseline is implemented with eager garbage collection, it should have had the lowest memory consumption. As its response time performance is low, some items need to stay in the system for a longer period of time, which explains why the second configuration behaves better. The lazy approach in garbage collection, the high block sizes determined by read patterns knowledge and also the data structure maintained for clustered reads slightly increased memory consumption for the third configuration. What these numbers suggest is that while the response time seriously improves, the changes in memory consumption are very low.

7.2 Effect of Fine Tuning

In the next experiments we conducted a more detailed analysis of the influence imposed by some parameter values on the performance. For this purpose, we generated a couple of scenarios using parts of the Linear Road implementation.

7.2.1 Even Finer Tuning

For the purpose of this experiment we simplified the "Accident Segments" query to only select the cars which have speed 0 on a per minute basis. This means that the query maintains a window on the car position reports it receives as input and accesses it in a sequential manner. For that we use an SMS Store Instance which should behave best for a FIFO update pattern and a Sequential read pattern (Figure 11), but we show results with different configurations as well (Random read and Clustered read patterns).

The input consists of over 6 million car position reports. Given different configurations of the Store, we measure how long it takes for the query to consume the input. We then divide the running time to the number of the items in the input to determine the throughput.

The results are presented in Figure 10. In this plot we can see that giving even slightly "wrong" configurations to the Store, the performance degrades. The most interesting one is probably the difference between having a Clustered and a Sequential read pattern: the Clustered implementation behaves like the sequential one

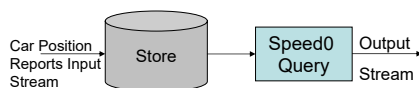


Figure 11: Speed 0 Experiment

	Shared	Not Shared
Average Memory Used	835MB	900MB
Average Throughput (tuples/s)	23,474	15,600

Table 7: Effect of sharing

as long as the requests fall in the same block. Whenever a request has to go to another block, instead of moving to the next one in the list, the Clustered implementation uses the id-index to locate the block. Using a Random pattern has such a low performance because of the continuous probing of the id-index with every request.

7.2.2 Effect of sharing

Next, we examine the effect of using the *Sharing* parameter. For this purpose, we designed a test scenario using two queries: the Speed0 query presented in the previous experiment and the "Segment Statistics for every minute" query (it also requires a one minute window). They both use as input the car position reports. *Not shared* is a test in which the input stream is generated twice and therefore a different store instance is created for each of the queries. In the *Shared* scenario, both queries connect to one single store instance (a "Shared" store) which materializes the input stream (Figure 12). The same number of input items was used and again we measured throughput and this time memory consumption as well. As expected, the *No share* setup consumes more memory (Table 7). Apart from higher memory consumption, the *Not Shared* setup had to do more computation: generate the input stream twice and materialize it into two storage instances. Therefore, it is no wonder that throughput was higher for the *Shared* scenario as opposed to the *No Share* one (Table 7). This experiment proves how important is for performance to allow and execute intermediate results sharing.

8. RELATED WORK

While there has been an extensive amount of recent work in the area of data stream management systems, the subject of stream data storage management has not seen much attention. Several research prototypes for DSMS have been built (e.g., Aurora [1], STREAM [15], TelegraphCQ [6]), all of which proposed different ways to deal with the above outlined storage requirements. However, all of these systems only support a limited set of the requirements. In addition, they have a tight coupling between query processing and storage management by embedding storage management within the stream processing engine, which is in contrast to the proposed separation of concerns in this paper. In this section, we briefly summarize the storage-related solutions that were proposed by the previous streaming systems.

Aurora uses two main forms of storage structures [1]: (i) tuple queues (FIFO, append-only) for storing intermediate results of continuous queries, and (ii) connection points as a persistent cache to store stream tuples for historical/ad-hoc queries as well as to plug in static tables for hybrid queries. Sharing is supported by allowing multiple operators to access the output queue of an upstream oper-

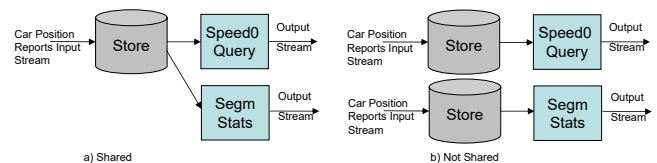


Figure 12: Effect of Sharing Setup

ator. If queues do not fit in main memory, the system spills them to disk selectively, based on the processing priorities of the operators.

STREAM also provides two basic storage structures [2]: (i) inter-operator queues to connect the operators, and (ii) synopsis to maintain operator state (also used for approximation). Sharing on tuple queues is supported in the same way as in Aurora. For synopsis sharing, STREAM provides a single physical store for the tuples, but multiple stubs (representing multiple views) for access. Queues and synopsis are allowed to overflow to disk in case of memory limitations, but no sophisticated algorithms are provided.

TelegraphCQ builds on several different types of dataflow modules, each with a different functionality [6]. A State Module (SteM) is used to temporarily store tuples for stateful operators (e.g., join), and supports build-probe-eviction operations as well as indexed and shared data access [16]. Furthermore, the Fjord API allows multiple modules to be connected via pull or push queues [14]. Finally, the PSoup extension on SteMs enables historical data access [7].

More recently, Golab et al have proposed storage techniques and index structures for sliding windows over data streams [11]. Basic windows are implemented as a linked list of tuples, while sliding windows are implemented as a circular array of pointers to these basic windows. On top of this, higher level storage structures based on attribute value aggregation are proposed. This work showed that indexing sliding windows over data streams can increase query processing efficiency. A follow-up to this work observed that different operators have different update patterns to their internal state, and this can be exploited for efficient query execution [12]. Query plans are annotated with update patterns of their constituent operators, and these annotations drive the use of different operator implementations as well as different data structures for state maintenance and result storage. Our SMS approach also exploits read patterns in addition to the update patterns. Furthermore, SMS is not concerned with using different operator implementations due to its decoupled design from the query processor.

9. CONCLUSIONS AND FUTURE WORK

This paper addresses the requirements of data-intensive stream processing applications. Such streaming applications do not only stress the processing engine of a DSMS, but put up new challenges for the management of data: large, quickly changing datasets need to be handled with low overhead in terms of memory and CPU while providing efficient access and low response times guarantees.

To address these challenges, we conducted a detailed requirements analysis. As a result of this analysis, we identified a key set of storage parameters together with the possible values that they should take under different situations.

The values of these parameters (in particular read and update patterns) provide the means to optimize stream storage to achieve the desired performance goals. We also describe their implementations based on state-of-the-art techniques.

While some of these implementations have already been used in an ad-hoc manner within existing DSMS, our work takes the next logical step and decouples storage management from the processing engine. Similar to traditional relational database systems, this approach provides flexibility, adaptation to specific requirements and optimizability. Our implementation of the storage manager, called SMS, provides a general-purpose storage system for DSMS and offers well-defined interfaces so that it can be tailored to different applications needs according to the storage parameters.

An experimental study of SMS on the Linear Road benchmark shows that significant improvements in query response time can be achieved by tuning the access pattern parameters differently for different stores involved in the benchmark. In the study, we also

analyze the effect of fine-tuning with the help of simpler test scenarios. The results obtained are very promising to prove that our decoupled design can greatly improve performance.

As future work, we would like to continue our investigation to determine whether our work applies to Complex Event Processing (CEP) engines as well. Another direction is to analyze the semantics of joining live and archived data and its impact on the storage management. Finally, we are planning to extend our techniques to distributed settings.

10. REFERENCES

- [1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal, Special Issue on Best Papers of VLDB 2002*, 12(2), 2003.
- [2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2007.
- [3] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, Toronto, Canada, September 2004.
- [4] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal Special Issue on Data Stream Processing*, 13(4), 2004.
- [5] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *VLDB Conference*, Vienna, Austria, September 2007.
- [6] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, Asilomar, CA, January 2003.
- [7] S. Chandrasekaran and M. J. Franklin. PSoup: A System for Streaming Queries over Streaming Data. *VLDB Journal, Special Issue on Best Papers of VLDB 2002*, 12(2), 2003.
- [8] H. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB Conference*, Stockholm, Sweden, August 1985.
- [9] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD Conference*, San Diego, CA, June 2003.
- [10] N. Gance, M. Hurst, and T. Tomokiyu. BlogPulse: Automated Trend Discovery for Weblogs. In *WWW Workshop on the Weblogging Ecosystem: Aggregation, Analysis and Dynamics*, New York, NY, May 2004.
- [11] L. Golab, S. Garg, and M. T. Özsu. On Indexing Sliding Windows over Online Data Streams. In *EDBT Conference*, Crete, Greece, March 2004.
- [12] L. Golab and M. T. Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.
- [13] R. Kuntschke, T. Scholl, S. Huber, A. Kemper, A. Reiser, H.-M. Adorf, G. Lemson, and W. Voges. Grid-based Data Stream Processing in e-Science. In *IEEE International Conference on e-Science and Grid Computing (eScience 2006)*, Amsterdam, The Netherlands, December 2006.
- [14] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *IEEE ICDE Conference*, San Jose, CA, February 2002.
- [15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, Asilomar, CA, January 2003.
- [16] V. Raman, A. Deshpande, and J. M. Hellerstein. Using State Modules for Adaptive Query Processing. In *IEEE ICDE Conference*, Bangalore, India, March 2003.