Towards Archiving-as-a-Service: A Distributed Index for the Cost-effective Access to Replicated Multi-Version Data

Filip-Martin Brinkmann and Heiko Schuldt
Databases and Information Systems Research Group
University of Basel, Switzerland
{filip.brinkmann | heiko.schuldt}@unibas.ch

ABSTRACT

With the advent of data Clouds that come with nearly unlimited storage capacity combined with low storage costs, the well-established update-in-place paradigm for data management is more and more replaced by a multi-version approach. Especially in a Cloud environment with several geographically distributed data centers that act as replica sites, this allows to keep old versions of data and thus to provide a rich set of read operations with different semantics (e.g., read most recent version, read version not older than, read data as of, etc.). A combination of multi-version data management, replication, and partitioning allows to redundantly store several or even all versions of data items without significantly impacting each single site. However, in order to avoid that single sites in such partially replicated data Clouds are overloaded when processing archive queries that access old versions, query optimization has to jointly consider version selection and load balancing (site selection). In this paper, we introduce ARCTIC, a novel cost-aware index for version and site selection for a broad range of query types including both fresh data and archive data. We describe in detail the interplay between the different parts of the index and their implementation. Moreover, we present the results of the evaluation of the combined version and replica index in a Cloud environment that shows a significant gain in query throughput compared to a monolithic index.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed databases; H.2.2 [**Physical Design**]: Access methods

Keywords: Data Archiving, Multi-version Data Management, Archiving-as-a-Service, Multi-version Index, Replication

IDEAS '15, July 13 - 15, 2015, Yokohama, Japan

http://dx.doi.org/10.1145/2790755.2790770.

1. INTRODUCTION

Cloud computing comes with the promise of providing literally unlimited resources, such as CPU cycles or storage capacity. Together with moderate prices and the pay-as-you-go model where only the resources that are actually used have to be paid for, this is a highly attractive environment for both business and private users. Instead of maintaining their own hardware for data management and/or data processing, they run their applications in the Cloud.

Data management in the Cloud is governed by the CAP theorem [6, 10]. In the presence of several geographically distributed data centers, most Cloud providers focus on a high degree of data availability by replicating data across sites and by lazily updating these replicas with the consequence that consistency is relaxed. Due to the lazy updates, several different versions per object might exist at each point in time at different sites. Hence, the well-established update-in-place paradigm for data management where old data are overwritten as soon as changes need to be stored is more and more replaced by a multi-version approach. In this context, keeping not just some but *all* versions of an object is a seamless extension of weak consistency protocols in a replicated data Cloud and can be rather cheaply realized due to the large available storage capacity combined with the low storage costs of the Cloud.

With partial replication applied to versions, the additional storage demands can be shared across sites in the Cloud without significantly impacting each single site. The only guarantee needed is that each version is available on a subset of sites. Hence, no single site needs to store all versions of all objects while, at the same time, the complete version history can be kept without a gap in the entire system. With multi-version data management in the Cloud, a much richer set of read operations with different semantics can be provided, compared to the read semantics that can be supported in a Cloud with update-in-place and eager replication. While the latter case only allows to access the most recent version of each object ("freshest" data), the former also supports archive queries such as "give me objects as of time t", "give me objects not older/younger than t", or "give me some/all version(s) between t_1 and t_2". We refer to an approach that treats such queries as first class citizens and that is able to jointly provide access to up-to-date data and to archived data in the Cloud as *Archiving-as-a-Service (AaaS)*.

When partial replication is used, several sites may qualify for the execution of these queries. In order to avoid that single sites are overloaded, query processing and optimization has to jointly consider version selection and load balancing

(site selection). An AaaS approach based on multi-version data management and partial replication thus has to address the following challenges: where can a particular data item be found, where are the versions of this data item located, which versions qualify for a given query, and which site is able to provide the version specified in a query in the cheapest way. The latter should consider the different capabilities of sites in a Cloud, such as available storage space and monetary costs for data storage, transfer costs, network latency, query times, etc. Hence, archive queries need to be jointly optimized across sites and versions.

In this paper, we introduce ARCTIC, a novel cost-aware index for version and site selection for a broad range of query types including both fresh data and archive data. We describe in detail the interplay between the different parts of the index and their implementation.

Consider, as an example for the use of ARCTIC, a globally distributed online stock trading website. Users can use it to look up and track stock quotes as well as buy and sell them. The website uses replicated and partitioned key/value (K/V) stores. The data in the K/V stores include stock descriptions, current and past stock quotes, and aggregated values such as, for instance, the average price per stock during the last 12 months. There are three different types of queries that account for more than 80% of the database load. *i.) What is a stock's description?* – the first query a user submits when looking up a stock. *ii.) What is the current quote of the stock?* – which accesses the most recent version. Finally, *iii.) How did the quote develop over a specific timespan?* – accesses all versions in a given interval and is essential for users to analyze stocks. With ARCTIC, users are able to seamlessly support all types of queries, i.e., a user should not have to switch between an online system and an archive in the course of her search.

The contribution of this paper is threefold. First, we introduce a rich query interface tailored to the AaaS environment which ARCTIC implements. This query interface includes traditional access to fresh data and in particular various types for archive data with different read semantics. Second, we show how cost-optimized access to partially replicated versions for these AaaS queries can be provided. This is achieved by the ARCTIC distributed replica and version index that selects the appropriate version(s) and chooses the most cost-effective replicas. Third, we provide an implementation of ARCTIC and an evaluation on the basis of amazon AWS Cloud resources. The evaluation results show a significant improvement in query throughput of the ARCTIC replica and version indexing compared to an integrated, monolithic indexing approach.

The remainder of this paper is organized as follows: Section 2 discusses related work. The system model is presented in Section 3. The ARCTIC index is introduced in Section 4 and its implementation and evaluation results are presented in Section 5. Section 6 concludes.

2. RELATED WORK

For decades, databases have been designed to apply updates in-place. This means that each update of an object overrides the previous version of this object. In order to correctly handle failures and aborts of transactions, databases use write-ahead logging (WAL) protocols such as ARIES [17] for correctly handling updates in-place. As an alternative to update in-place, so-called shadow page approaches are used where each object exists in two versions (a valid one and one to which an active transaction is applying updates; with the commit of the update transaction, the later becomes the new valid version).

Multi-version databases extend the shadow page approach by keeping the complete version history of each object. The term multi-version database was coined back in the early 1980s, when the concept of multi-version concurrency control started to attract increased interest [4, 5]. An early overview on the different access structures and methods for multi-version data is provided in [16]. In general, one can distinguish between tree-based and hashing indexes. Tree-based indexes allow for range queries (both key and version ranges) while hashes are mostly used for exact and membership queries.

Temporal databases explicitly consider the time dimension together with data. Research on temporal databases [23, 12] includes temporal extensions to SQL [24] which have been incorporated into SQL:2011 [15]. The definition of the bi-temporal model [13] provides a solid foundation for various works regarding temporal databases. This bi-temporal model distinguishes between valid time (i.e., the time in which an object has been valid in the real world) and transaction time (i.e., the time in which the object stored in the database was considered true). An approach to structure and distribute access methods for temporal/multi-version data is described in [18]. The Log-structured History Access Method LHAM is a temporal extension of the log-structured merge tree LSM [19]. It assumes several horizontally ordered, independent index structures. The topmost index structure will sit on the fastest memory in the system, while the one below will use a slower, but larger storage. The basic idea is to use rolling merges for transferring entries from a higher to a lower index structure. Another approach is to distribute the inner and leaf nodes of a B^+ tree between arbitrary nodes [1, 25]. While this approach is more fine granular, it needs (mini-)transactions in order to change the distributed tree structure.

In distributed systems, lazy replication and thus weak consistency following the BASE semantics (Basically Available, Soft state, Eventual consistency) [20] is more and more replacing the strong ACID semantics (Atomicity, Consistency, Isolation, Durability) [11] of transactions. The lazy propagation of updates to replicas also leads to different versions of objects and thus to stale data (i.e., versions that are no longer valid) [7]. It can further be used for trading freshness against lower replication cost [21, 2]. As a consequence, richer read semantics can be offered by operations like *readNotOlderThan*, leading to improved system performance while offering freshness guarantees [27].

3. THE ARCTIC SYSTEM MODEL

ARCTIC combines multi-version data management and partial replication across sites in a Cloud environment. In ARCTIC, we build on recent approaches to Cloud data management with weak consistency and consequently keep not just some version with stale data but *all* versions, at least one copy of each version at some site in a distributed system (optionally also replicas of it on other sites in order to increase the level of availability), to seamlessly combine Cloud data management and data archiving.

The ARCTIC system consists of n sites and hosts m data objects. For each object x_p, the complete version history

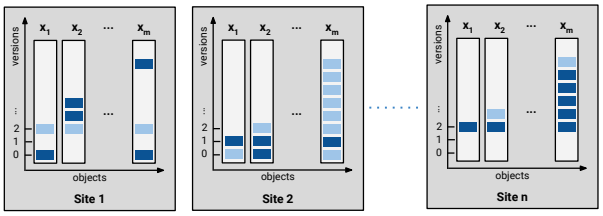

Figure 1: Partial Replication in ARCTIC

is stored. Each version is associated with a version number; for each object, the version numbers are assigned in strongly monotonically increasing order. In what follows, the object ID (whenever necessary) is used as subscript, the version number as superscript (i.e., x_p^i is the i^{th} version of object x_p). At the same time, each version x_p^i of object x_p is associated with an interval $[t_{p,s}^i, t_{p,e}^i[$ in which this version has been valid in the system. In here, $t_{p,s}^i$ is the time at which version x_p^i has been created, and $t_{p,e}^i$ is the timestamp at which version x_p^i has become outdated as the next version x_p^{i+1} of object x_p has been produced ($t_{p,e}^i = t_{p,s}^{i+1}$). Hence, given a timestamp t, the valid version j of an object x_p can be identified by $t_{p,s}^j \leq t < t_{p,e}^j$.

With $x_p^i[l]$ we denote a replica of version x_p^i at site l. In ARCTIC, for each object, each version is stored at least at one site; in order to increase availability and to allow for optimizing read accesses, usually the replication degree r_p^i for x_p^i s is much higher ($1 \leq r_p^i \leq n$ for all objects x_p and all its versions x_p^i). However, r_p^i might differ for different versions of the same object and for different objects. This allows the system to dynamically decide on the best number of replicas depending on the access pattern.

Figure 1 shows an overview of the ARCTIC system. The distributed sites are symbolized by the gray boxes. Each site stores a partition of the set of versions of all objects in the form of local replicas. The versions are depicted as boxes inside the objects $x_1, \ldots, x_m$. Each version is assigned to a specific site, which serves as its master copy. This is the replica site at which this version has been created. Replicas are inserted at the master, but can be replicated to any other site as well. The master copies are represented by the dark boxes, while further copies are shown as light boxes. As described above, each version is stored at least once in the form of a replica. Each site contains the same logic and can be queried by clients. While insert and update operations must take place at the master, read access is possible at any site.

Since ARCTIC keeps all versions of all objects, it can offer a broad spectrum of query types that take into account the version number and/or the timestamp at which a particular version has been valid. Hence, queries can be thought of as arbitrarily shaped areas in the two-dimensional key/version creation time space. Consider Figure 2 which depicts various temporal queries. At some point in time t, a data item may or may not exist in the database. In Figure 2, we assume that all items exist at all times depicted, i.e., the object identified by key k exists at time t. A query asking for exactly this key at exactly this time is denoted as *read(k, t)* and depicted as point x_k^0 in Figure 2. However, not only *exact* queries are possible, but also queries specifying intervals,

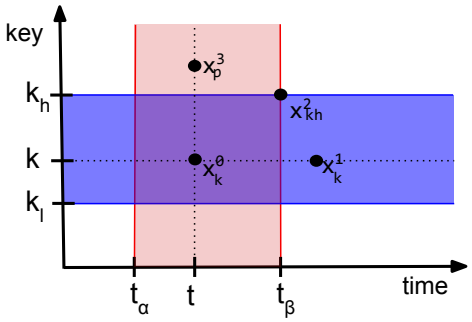

Figure 2: Key/Version Creation Time Queries

as mentioned earlier. Generally, all combinations of exact matches and intervals are possible. Table 1 summarizes the different query types supported by ARCTIC.

For both creation time t and key k, either a discrete value or an interval can be used when specifying a query $read(k, t)$. Intervals may be bounded or unbounded. If an interval boundary is given, it might be excluded or included. In Table 1, for reasons of simplicity, we have just provided examples in which the boundaries are excluded (for instance R(k, $t_\alpha < t < t_\beta$) asks for all versions of t older than t_β and younger than t_α). However, R(k, $t_\alpha \leq t \leq t_\beta$) is possible as well. In this case, the semantics would be "give me all versions of t as of t_α or younger and as of t_β or older". Moreover, bounds can also be omitted (which means that either all valid times or all keys need to be considered), or intervals may be half-bounded, i.e., only an upper or lower bound in the key range or time interval is given. In addition, for version selection, either a timestamp (time interval) can be specified (as shown in Table 1), or a dedicated version number (a version number interval, respectively) can be specified. Hence, each of the query types listed in Table 1 comes in two different signatures – one with the specification of valid time, one with the specification of the version(s) – yet both have the same semantics.

However, we further allow each query to include at least one (possibly unbounded) time interval to be satisfied with the return of *only one* version. This leads to an important class of queries which allow for accessing possibly *stale* versions of data items by relaxing *freshness* requirements on the result set. We use uppercase read operations, R(k, t), for queries which are supposed to return *all* versions that qualify for the specified key and/or time intervals, and lowercase read operations, r(k, t), for reads that are satisfied with *at least one* version from the specified interval. Previous work has shown that using freshness of data items as a metric can

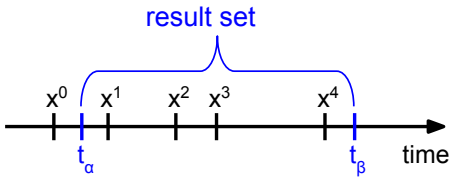

Figure 3: Freshness Query

Table 1: Version Queries in ARCTIC

Query	Semantics	Where shown in Figure 2
Discrete values		
R(k, t)	Returns a version of data item with key k which has been valid at time t or ...	x_k^0
R(k, v)	... that has version number v.	
Bounded intervals and discrete values		
R($k_l < k < k_h$, t)	Returns all data items with keys between k_l and k_h and valid time t (*range-timeslice* [22]).	Vertical dotted line inside the rectangle between (k_l, t_α) and (k_h, t_β), e.g., x_k^0.
R(k, $t_\alpha < t < t_\beta$)	Returns all data item versions with key k and valid in the interval $]t_\alpha, t_\beta[$.	Horizontal dotted line inside the rectangle between (k_l, t_α) and (k_h, t_β), e.g., x_k^0.
Bounded intervals		
R($k_l < k < k_h$, $t_\alpha < t < t_\beta$)	Returns all data item versions with keys between k_l and k_h being valid between t_α and t_β.	The rectangle between (k_l, t_α) and (k_h, t_β), excluding the borders, e.g., x_k^0, but not x_{kh}^2.
Unbounded intervals		
R(–, $t_\alpha < t < t_\beta$)	Returns all data item versions with any key being valid between t_α and t_β.	The vertical, light red area excluding its borders, e.g., x_k^0 and x_p^3.
R($k_l < k < k_h$, –)	Returns all data item versions with keys between k_l and k_h and any valid time.	The horizontal, dark blue area excluding its borders, e.g., x_k^0 and x_k^1.
Unbounded intervals and discrete values		
R(k, –)	Returns the full version history of data item x with key k (*pure-key* in [22]).	$\{x_k^0, x_k^1\}$
R(–, t)	Returns all versions of all data items that have been valid at time t (known as *pure-timeslice* [22]).	$\{x_k^0, x_p^3\}$

be used to reduce cost of lazy replication [21, 27]. Consider a set of versions of one data item as depicted in Figure 3, in which a query asks for a version of data item x with key k in the time interval $[t_\alpha, t_\beta[$. If the read semantics is "give me all versions in this interval", i.e., R(k, $t_\alpha \leq t < t_\beta$), the result set of a complete query therefore is $\{x^1, x^2, x^3, x^4\}$. When the more relaxed query semantics is used "give me at least one version from this interval", i.e., r(k, $t_\alpha \leq t < t_\beta$), the requirements on the system are relaxed so that it can return any result set which meets the freshness requirement. Let us assume that the system first finds version x^2. The query execution time would be shorter if the system could just return version x^2, assuming that this answer is sufficiently fresh for the client, thus meets her requirements. Therefore, any query in Table 1 which contains a (half-)bounded time interval can be relaxed in this way.

Query optimization thus needs to fulfill the freshness requirements specified by a user. However, optimization may consider the selection of one or several versions in the specified interval and the selection of the replica(s) which materialize these versions.

4. ARCTIC: COST-AWARE MULTI-VERSION REPLICA INDEX

All the different types of version queries summarized in Table 1 have in common that the following two tasks need to be jointly supported:

1. The queries supported by ARCTIC ask for keys, versions/valid time intervals, or both. Thus, ARCTIC must be able to answer the question *"Which version represents a specific data item at a specific time or with a given version number?"*, i.e., to select one or several versions, according to the specification in the query.

2. Since every version can be materialized through multiple replicas, the site (or sites) that can best serve this/these version(s) need to be selected. This is important since we do not consider full replication, hence not each site is able to process a query locally.

A major feature of ARCTIC is to address both tasks separately, albeit not independently. This leads to a modular index structure that consists of two parts: First, a *version index* that maintains information on the different versions of each object, their version numbers and the intervals in which they have been valid. Second, a *replica index* that maintains information on the currently available replication state of versions. This knowledge is potentially highly volatile, especially when load balancing takes place and new replicas are created and/or existing replicas are removed from the system.

In addition, since we target large Cloud environments with a large number of data objects and in particular many versions per data object, especially the replication index may grow significantly over time. In order to find a good balance

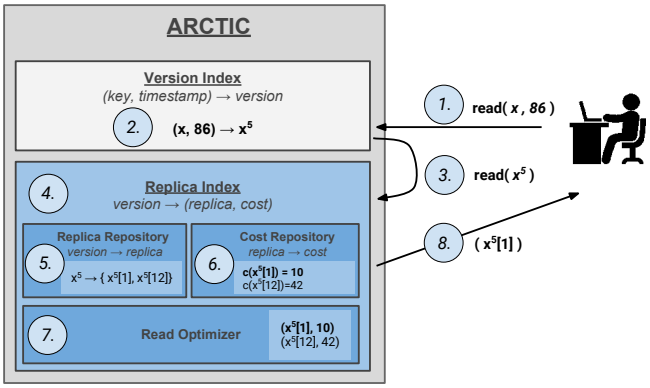

Figure 4: ARCTIC Query Execution

between a low maintenance overhead of the replica index and good performance, the replica index should neither be kept centrally as this would limit the scalability of the entire system, nor should it be replicated across all sites as this would create significant effort for keeping all replicas of the replica index consistent. Therefore, it is important that the replica index can be partitioned in such a way that neither its maintenance nor its usage are negatively affected.

4.1 ARCTIC Overview

The execution of a version query in ARCTIC can be broken down into the following steps: First, given a query containing key and time values or intervals, determine *which versions* form the result set of the query. Second, for each version, determine the *access costs*. Costs are not restricted to query latency only, but can include anything that needs to be optimized for, like network traffic, system load, etc. The costs depend on the set of actual replicas that are used to compose the result set. Each replica may have different costs. Third, among all strategies, each consisting of a set of replicas, the one with *minimal total costs* is selected and returned.

ARCTIC actually consists of two separate data structures, as illustrated in Figure 4. Consider, as an example, a query to the ARCTIC index that asks for data item x at timestamp 86, depicted in the upper right corner (1.). The ARCTIC *version index* (upper box) holds a mapping from keys and timestamps/version numbers to versions. In the example, the identifier x^5 stands for the version needed for answering the query *read(x, 86)* (2.). Note that in the case of range queries the answer would be a version set.

The ARCTIC *replica index* (lower large box) is used to find the cheapest replica which materializes version x^5 (3., 4.). The index consists of the following structures. A *replica repository* (5.) knows all replicas that materialize specific versions. In the example, these are the replicas $\{x^5[1], x^5[12]\}$, i.e., replicas of of version x^5 at the sites 1 and 12. The replica repository is partitioned among sites, for instance by using a distributed hashtable, which is what we use in our implementation. Each replica is associated with a cost that incurs upon its materialization. This mapping is captured by a *cost repository* (6.), which returns the costs for each of the two replicas. In the example, the calculated costs are $c(x^5[1]) = 10$ and $c(x^5[12]) = 42$. The third component, the *read optimizer* (7.), is tightly integrated with the two repositories. Its task is to yield the final cost-minimized result set. In the example, replica $x^5[1]$ yields the lowest cost (10) and is thus returned (8.).

Figure 5 shows how the two index structures are integrated into ARCTIC. Each site has local materializations of both index structures. While a complete copy of the version index is kept on every site, the replica index is partitioned across all sites. As mentioned earlier, the replica index contains globally unique detailed information about the replicas. Each site, however, keeps local information about its connectivity to other relevant sites. Currently, this information is lazily updated for every other site. However, it is possible to use heuristics or a centralized metadata repository in order to relieve a site from the burden of keeping track of the other sites.

4.2 Version Index

The version index needs to answer the queries in which a version number or a timestamp is specified. Any known index structure implementation can be used, since this index is independent of the replica index. Possibilities include, but are not limited to, well known index structures like the Multiversion B-Tree (MVBT) [3], the Snapshot Index [26] or the Time Index [8].

It should, however, be noted that since ARCTIC addresses archiving applications which require very long-running systems, the index structure must be suitable for horizontal partitioning such that old or less frequently queried version entries can be pushed to slower storage and/or vertical partitioning for load balancing between sites, for instance according to the LHAM [18] index. The actual partitioning scheme that is used depends on the read/write characteristics of the workload. Currently, we use a distributed hashmap which allows for horizontal partitioning while maintaining scalability. The index can, however, be easily replaced by any of the aforementioned index structures.

It is important to note that instead of creating a single index structure, we keep the version and the replica index separated due to the following reasons.

Separation of Concepts: The logical data schema (which contains versions) is separated from the physical representation. If both indices were represented by a single data structure, physical changes in the network would be reflected in the index, even if no changes to the version set were made. We assume that this separation results in an improved runtime behaviour of the index, since maintenance of the version index is only necessary when versions are added.

Improved Partitioning: Two index structures allow for more flexibility in partitioning. Different partitioning schemes can be applied to the two indices. In our current implementation, for example, the version index is not partitioned while the replica index is.

Global vs. Location-dependent Knowledge: On one hand, the contents of the version index are globally valid. On the other hand, some entries in the replication index are location-dependent. Moreover, the replica index is much more volatile than the version index, as we describe in the following section.

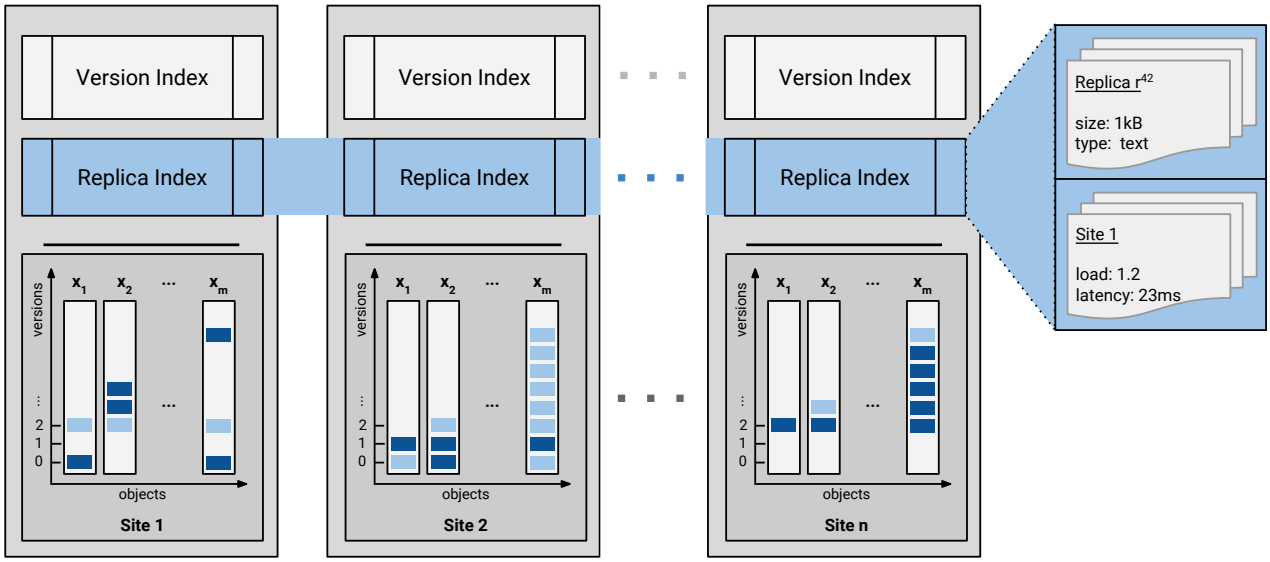

Figure 5: Integration of Indexes into ARCTIC

4.3 Replica Index

The replica index contains information about replicas of versions along with information which allows for calculating (or at least estimating) their cost. Replicas are spatially separated copies of the same version. When accessing a replica, different costs incur. Types of costs include time, network traffic generated, processing cost, storage costs, etc. The replica index takes as an argument the identifier of a version and returns the replica with the lowest cost or the replica set (including operations) which allows for materializing the desired version. Therefore, the mapping which the replica index provides is $version \mapsto \{(replica,\ cost)^*\}$.

The index must have access to the full set of replicas including their individual costs. It is the responsibility of the replica control mechanism to insert, delete and update entries in the replica list. The costs associated with each replica can be computed when the following three types of information are known: *replica metrics*, *site metrics* and *inter-site metrics*.

Replica Metrics: Inserting, storing and retrieving a replica creates costs. In order to compute these costs, specific metrics of each replica must be taken into account, e.g., size, computational complexity, availability, etc.

Site Metrics: The aforementioned replica metrics are not sufficient to compute the costs for managing replicas. The costs occurring on a highly frequented site may be higher than on an idle site. Therefore, site metrics describe site-specific costs. Examples include load, (remaining) storage space, types of storage space (main memory, disks, tapes), uptime costs, etc. Metrics are regularly exchanged between sites. A centralized site metric repository would be also possible.

Inter-Site Metrics: The third type of metrics captures the connection details of sites – more concretely the paths between nodes. It is potentially much more dynamic than the former two types and as well larger. This is due to the fact that pairwise metrics are needed, thus giving it a size of $\mathcal{O}(N^2)$ with N being the number of sites. This information is collected by a regularly scheduled bandwidth and latency test between sites.

The replica index consists of three parts. First, the *replica repository* contains the information about all active replicas and the mapping between versions and replicas. Since replicas can be identified by ordered keys, a distributed hashmap can be used as a data structure, partitioning its information among all sites. It also holds further metadata about replicas, like for instance their size, compression, etc. The *cost repository* can be used to look up further cost factors for each replica, i.e., information and heuristics about the participating sites holding the replicas and their interconnections. Since the information in the cost repository differs between sites, each site maintains its own version of the cost repository. The joint information of cost and replica repository is then used by the *read optimizer* to return the optimal solution to the version read problem.

4.4 Query Execution in ARCTIC

Each ARCTIC site is able to answer read queries. Upon arrival of a query, the site's local version index is used to determine the set of versions that are able to satisfy the query. For each version, one site holds the partition of the replica index which can be used to identify the sites which hold the version's replicas. Currently, ARCTIC directly calculates the sites with a hash function executed on the version; however, this can be easily exchanged with a more adaptable index structure, if necessary. From the set of resolved replicas and the sites they are stored on, the cheapest strategy is calculated. This is achieved by taking the replicas' cost information like, for instance, size into account as well as the sites' current load. When a strategy which satisfies the query is found, the resulting replica set is downloaded from the site(s) and returned to the client.

5. ARCTIC IMPLEMENTATION AND EVALUATION

We implemented ARCTIC on top of PolarDBMS [9], a Cloud database framework which is currently being developed at the University of Basel. Since we are interested in the read performance, we focused our evaluations on the query response time characteristics of the system.

The dataset we used consists of 46,028 data objects of which in total 134,574 versions have been created. The temporal distribution of the dataset was given, since we used a pre-generated TPC-BiH dataset [14] in order to create the key/value pairs representing the database objects.

5.1 Evaluation Set-up

We deployed the application to eight Amazon Web Service EC2 *m3.medium* instances in the same region *eu-central-1*. All experiments were set up as follows: we defined a query mix QM consisting of the three query types i.) *read most recent*, ii.) *read as of* and iii.) *read not older than*. The queries that were considered in the evaluation were selected with a uniform distribution from these query types.

We created 4,000 queries asking for random existing keys in the key range. For the *read as of* and *read not older than* queries, we selected random timestamps from the data item's valid time. We created three replicas of each data item version with a size of 10 kB, thus leading to a total dataset size of 403,722 replicas. The replicas were distributed over all system instances such that each version was available on three instances.

The general setup was as follows. One of the instances, which we call the primary instance, is dedicated to answering queries. All queries were issued from a client application on this instance. The instance measured the time between arrival of a query and the moment in which it could deliver the result. Therefore, the client application speed did not affect the overall performance.

In order to evaluate the impact of the ARCTIC index, we conducted three successive experiments. In the first, which serves as a baseline, no index is available. The second setup allows for each instance to use the replica index to choose an instance on which the data item version is available — however, no further optimization is applied (e.g., the index does not allow to select the least loaded node hosting a replica of the data to be accessed). The third setup uses the complete ARCTIC index and thus also allows for cost-based optimization as the system is able to pick the cheapest replica.

Each experiment, consisting of 4,000 queries, was executed three times in order to avoid anomalies stemming from the underlying infrastructure.

Since the inter-site latencies are very low (between 0.5 and 3 ms) within the same AWS region, we added artificial latencies of 20ms to four of the eight nodes. This enables us to introduce the asymmetry that exists when spreading data to several data centers. However, since latency variance is very unpredictable in our experience, we have chosen this approach in order to be able to model inter-data center characteristics more precisely and predictably.

Latencies were determined by a process on each site which regularly pinged all other nodes and computed the average of the last three ping latencies. The process measured latencies every 60 seconds.

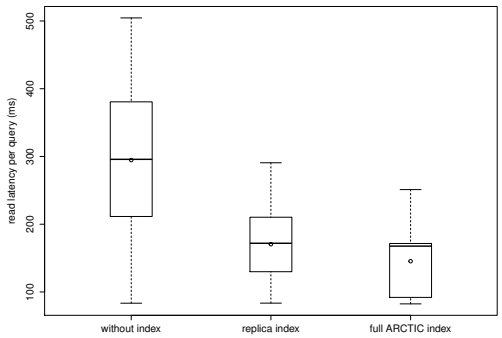

Figure 6: 4,000 Queries across five simulated data centers

5.2 Contribution of the different Indexes to the ARCTIC Performance

In what follows, we present a series of evaluations that assesses the contributions of the different parts of the ARCTIC index to the overall optimization of query execution time for three types of archive queries.

5.2.1 Baseline Evaluation

When a query comes in, the primary instance translates the desired key and version combination into a list of replicas that can potentially answer the query. If it does not hold any of the replicas itself, it picks one of the other sites at random which will then be queried. If it can answer the query, it will deliver its replica. If not, the next random site among the remaining sites is selected and queried. This procedure is repeated until the query is successfully answered. The query response time thus not only contains the time needed for the delivery of the replica but also the (potentially unsuccessful) requests submitted to other sites. Figure 6 shows the query answer times for all three experiments (baseline, replica index only, ARCTIC replica and cost index). The query execution times of the baseline evaluation are summarized by the leftmost whisker box.

The horizontal line inside each box shows the median, while the upper and lower box limits depict the upper and lower quartile of all measurements. The dot inside each box shows the mean value.

In addition, Table 2 provides the exact values for both the mean and the standard deviation of the query response times. As one can see, the average query time of the baseline is around 294.65 ms. 50% of the queries are answered within approx. 210 to 390 ms. However, approximately 25% of the queries exceed 390 ms. In all three experiments, the minimal access time was 82-83 ms which is dominated by the download time.

Table 2: Query Latency Times (ms) for 4,000 Queries

	mean (stdev)
No index	294.65 (11.43)
Replica Index	170.51 (54.17)
Full Index	145.44 (45.87)

Table 3: Query Latency Times (ms) for 4,000 Queries, 4 Regions

	Low Load: mean (stdev)	High Load: mean (stdev)
Replica Index	711.74 (418.42)	861.68 (587.81)
Full Index	594.12 (439.62)	609.40 (473.36)

5.2.2 Execution Times with Replica Index

For the second experiment, we enabled the replica index, yet without the cost repository. This allows for the primary instance to create a list of instances which are able to answer the incoming query. From this list, a random site is chosen. This leads to the decreased query execution times as seen in Figure 6. All queries can now be answered with at most one inter-instance communication — but without the possibility to select the cheapest among these replicas. The primary instance trades the time that is needed for consecutively trying the other nodes until a matching replica is found for a much smaller time needed for a lookup in the local replica index. The mean access time is reduced from 294.65 ms to 170.51 ms.

5.2.3 Execution Times with Full ARCTIC Index

In the final experiment, we enabled the full ARCTIC index, thus allowing for the system to select the cheapest replica representing the queried version. The costs were calculated by equally weighting the latency and the bandwidth of the site holding the desired replica. In our setup, the instances vary in their latency while the bandwidth of each instance is the same.

The rightmost whisker box in Figure 6 shows how the overall access times have improved. The mean access latency has decreased by approx. 15% to 145.44 ms. This shows that the read optimizer improves the overall access performance, leading to decreased mean and maximum access latencies.

5.3 Performance Evaluation in Multi-Data Center Configuration

In order to measure the impact of the cost index in heterogeneous environments, we deployed the system across four AWS regions: *eu-central-1* (Frankfurt, Germany), *us-east-1* (N. Virginia, USA), *us-west-1* (N. California, USA) and *ap-southeast-2* (Sydney, Australia). In each region, we deployed four instances with the same index configuration and the same dataset as in the first evaluation series. We selected four master instances which performed the queries such that each region contains exactly one master. The same query mix of 4,000 random queries was chosen. In order to test the performance of the cost index, we executed a script which selected one of the non-master instances every 60 seconds. This instance's network connection and general I/O was then set under heavy load by downloading and storing a randomly generated byte stream for approx. one minute from a machine in the same region. We conducted the experiment in four different configurations. First, we disabled the cost optimization. This setup was then evaluated under low load and then under high load by executing the aforementioned script. The same experiment was then repeated with the cost optimization being enabled, thus giving the system access to the full ARCTIC index. It has to be taken into account that query latencies are, due to the global setup, much higher than in the previous evaluations.

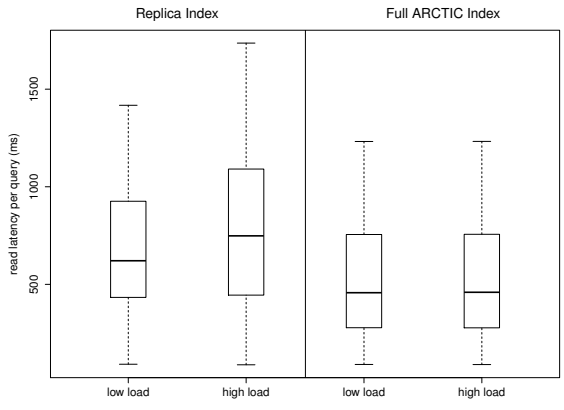

Figure 7: 4,000 queries across four regions

Table 3 shows the average query times for the four possible configurations. Figure 6 shows the box plots of the evaluation results. Two observations can be made. First, the results of the low load scenario show that when cost optimization is active, query times are significantly lower. This shows that in a static environment with different replica costs (e.g., different site latencies and bandwidths), the index allows for reducing the access costs. The second observation can be made when comparing the two scenarios "Replica Index" and "Full ARCTIC Index". It can be seen that when the system is set under load, query latencies in the first scenario increase significantly. In the second scenario with the activated cost optimization, query latencies stay nearly constant. This demonstrates that the ARCTIC cost optimization works in highly distributed and dynamic environments.

6. CONCLUSION

We have introduced ARCTIC, a novel index structure that is tailored to archive queries in an Archiving-as-a-Service (AaaS) context in which users may search for different (possibly outdated) versions of a data item in a partially replicated data Cloud. ARCTIC consists of a version index to identify one or several versions of a data object specified by a user, and a replica index which finds the cheapest replica of these versions in the system. ARCTIC makes use of the vast number of low-cost storage resources available in the Cloud by keeping all versions of all data objects and by applying partial replication of these versions across sites (data centers) in a data Cloud. This multi-version data management approach is a seamless extension of current practices in data Clouds where due to lazy propagation of updates in the presence of replication, several versions of objects exist. The evaluations of ARCTIC have shown the contribution of the different parts of the ARCTIC index to the optimization of archive queries. Moreover, ARCTIC has demonstrated that it selects the cheapest replica even in situations of increased system load and thus significantly reduces query latency.

In our future work, we will further investigate the effects of heterogeneous sites (e.g., network latency) on the query execution time with ARCTIC. Moreover, we will also add

support for incremental version management, i.e., to allow for some versions only to store the delta compared to a previous version. Especially for large data items, this will allow to significantly reduce the volume of storage needed (which has to be paid for in the Cloud), but it poses additional challenges for the execution and optimization of a query (e.g., local re-creation of a version via deltas vs. remote access to the materialized version). Since ARCTIC particularly aims at supporting large Cloud data management infrastructures, we will evaluate its performance when dealing with increasing amounts of sites, data objects, versions and replicas. This will allow for us to tune its parts, like for instance version and replica index, in order to ensure performance while maintaining scalability.

7. REFERENCES

[1] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-Tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.

[2] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 565–576, Trondheim, Norway, 2005. ACM.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, Dec. 1996.

[4] P. A. Bernstein and N. Goodman. Concurrency control algorithms for multiversion database systems. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing*, PODC '82, pages 209–215. ACM, 1982.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[6] E. A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*, 2000.

[7] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *ACM Sigmod Record*, volume 29, pages 117–128. ACM, 2000.

[8] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 1–12, San Francisco, CA, USA, 1990.

[9] I. Fetai, F. Brinkmann, and H. Schuldt. PolarDBMS: Towards a cost-effective and policy-based data management in the Cloud. In *Workshops Proc. of the 30th International Conference on Data Engineering Workshops*, pages 170–177, 2014.

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[11] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Compututing Surveys*, 15(4):287–317, 1983.

[12] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Rec.*, 21(3):35–43, Sept. 1992.

[13] C. S. Jensen and R. T. Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, June 1996.

[14] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. TPC-BiH: A Benchmark for Bitemporal Databases. In *Proceedings of the 5th TPC Technology Conference on Performance Characterization and Benchmarking (TPCTC 2013)*, pages 16–31, Trento, Italy, Aug. 2013.

[15] K. Kulkarni and J.-E. Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.

[16] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 315–324, New York, NY, USA, 1989. ACM.

[17] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[18] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, Feb. 2000.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[20] D. Pritchett. BASE: An ACID Alternative. *Queue*, 6(3):48–55, May 2008.

[21] U. Röhm, K. Böhm, H. Schek, and H. Schuldt. FAS - A freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 754–765, Hong Kong, China, Aug. 2002. Morgan Kaufmann.

[22] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.

[23] R. T. Snodgrass. Temporal databases. *IEEE Computer*, 19:35–42, 1986.

[24] R. T. Snodgrass. *The TSQL2 temporal query language*, volume 330. Springer Science & Business Media, 1995.

[25] B. Sowell, W. Golab, and M. A. Shah. Minuet: a scalable distributed multiversion b-tree. *Proceedings of the VLDB Endowment*, 5(9):884–895, 2012.

[26] V. J. Tsotras and N. Kangelaris. The snapshot index: An i/o-optimal access method for timeslice queries. *Information Systems*, 20(3):237–260, May 1995.

[27] L. Voicu, H. Schuldt, Y. Breitbart, and H.-J. Schek. Flexible data access in a cloud based on freshness requirements. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 180–187, 2010.