Cheap Data Analytics using Cold Storage Devices

Renata Borovica-Gajić[⋆]
renata.borovica@epfl.ch

Raja Appuswamy[⋆]
raja.appuswamy@epfl.ch

Anastasia Ailamaki[⋆] [‡]
anastasia.ailamaki@epfl.ch

[⋆]École Polytechnique Fédérale de Lausanne [‡]RAW Labs SA

ABSTRACT

Enterprise databases use storage tiering to lower capital and operational expenses. In such a setting, data waterfalls from an SSD-based high-performance tier when it is "hot" (frequently accessed) to a disk-based capacity tier and finally to a tape-based archival tier when "cold" (rarely accessed). To address the unprecedented growth in the amount of cold data, hardware vendors introduced new devices named *Cold Storage Devices (CSD)* explicitly targeted at cold data workloads. With access latencies in tens of seconds and cost/GB as low as $0.01/GB/month, CSD provide a middle ground between the low-latency (ms), high-cost, HDD-based *capacity* tier, and high-latency (min to h), low-cost, tape-based, *archival* tier.

Driven by the price/performance aspect of CSD, this paper makes a case for using CSD as a replacement for both capacity and archival tiers of enterprise databases. Although CSD offer major cost savings, we show that current database systems can suffer from severe performance drop when CSD are used as a replacement for HDD due to the mismatch between design assumptions made by the query execution engine and actual storage characteristics of the CSD. We then build a CSD-driven query execution framework, called Skipper, that modifies both the database execution engine and CSD scheduling algorithms to be aware of each other. Using results from our implementation of the architecture based on PostgreSQL and OpenStack Swift, we show that Skipper is capable of completely masking the high latency overhead of CSD, thereby opening up CSD for wider adoption as a storage tier for cheap data analytics over cold data.

1. INTRODUCTION

Driven by the desire to extract insights out of data, businesses have started aggregating vast amounts of data from diverse data sources. Emerging application domains, like Internet-of-Things, are expected to exacerbate this trend further [18]. As data stored in analytical databases continues to grow in size, it is inevitable that a significant fraction of this data will be infrequently accessed. Recent analyst reports claim that only 10-20% of data stored is actively accessed with the remaining 80% being *cold*. In addition, cold data has been identified as the fastest growing storage segment, with a 60% cumulative annual growth rate [17, 19, 42].

As the amount of cold data increases, enterprise customers are increasingly looking for more cost-efficient ways to store data. A recent report from IDC emphasized the need for such low-cost storage by stating that only 0.5% of potential Big Data is being analyzed, and in order to benefit from unrealized value extraction, infrastructure support is needed to store large volumes of data, over long time duration, at extremely low cost [17].

To reduce capital and operational expenses when storing large amounts of data, enterprise databases have for decades used storage tiering techniques. A typical three-tier storage hierarchy uses SSD/DRAM to build a *low-latency* performance tier, SATA HDD to build a *high-density* capacity tier, and tape libraries to build a *low-cost* archival tier [42].

Considering the cold data proliferation, an obvious approach for saving cost is to store it in the archival tier. Despite the cost savings, this is unfeasible due to the fact that the tape-based archival tier is several orders of magnitude slower than even the HDD-based capacity tier. As enterprises need to be able to run batch analytics over cold data to derive insights [18], the minute-long access latency of tape libraries makes the archival tier unsuitable as a storage medium for housing cold data.

In the light of limitations faced by the archival tier, storage hardware vendors and researchers have started explicitly designing and developing storage devices targeted at cold data workloads [36, 40, 41]. These devices, also referred as *Cold Storage Devices (CSD)*, pack thousands of cheap, archival-grade, high-density HDD in a single storage rack to achieve very high capacities (5-10PB per rack). The disks are organized in a Massive-Array-of-Idle-Disks (MAID) configuration that keeps only a fraction of HDD powered up at any given time [7].

CSD form a perfect middle ground between the HDD-based capacity tier and tape-based archival tier. Due to the use of cheap, commodity HDD and power-reduction provided by the MAID technology, they are touted to offer cost/GB comparable to the traditional tape-based archival tier. For instance, Spectra's ArticBlue CSD is reported to reduce storage cost to $0.1/GB [41], while Storiant claims a total cost of ownership (TCO) as low as $0.01/GB per month [29]. Due to the use of HDD instead of tape, they reduce the worst-case access latency from minutes to mere seconds–the spin up time of disk drives. Thus, performance-wise, CSD are closer to the HDD-based capacity tier than the archival tier.

Motivated by the price aspects of CSD, in this paper we examine *how CSD should be integrated into the database tiering hierarchy*. In answering this question, we first make the case for modifying the traditional storage hierarchy by adding an entirely new tier referred to as the *Cold Storage Tier* (CST). We show that enterprises can save hundreds to millions of dollars by using CST to replace both HDD-based capacity and tape-based archival tiers.

We then present an investigation of the performance implications of using CSD as a replacement for the traditional capacity tier of

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

enterprise databases. We first show that current database systems can suffer from severe performance hit when CSD are used as a replacement for the capacity tier due to the mismatch between design assumptions made by the query execution engine and actual storage characteristics of the CSD.

Then, we introduce Skipper – a new CSD-targeted, query execution framework that modifies both the database execution engine and CSD scheduling algorithm to be aware of each other and operate towards achieving a common goal–masking the high access latency of CSD. In particular, Skipper employs an adaptive, CSD-driven query execution model based on multi-way joins which are tailored for out-of-order data arrival. A detailed evaluation shows that this execution model coupled with efficient cache management and CSD I/O scheduling policies can mask the high latency overhead of CSD and provide substantially better performance and scalability compared to the traditional database architecture.

Our contributions are as follows:

- A cost and performance analysis which demonstrates that the CSD-based cold storage tier can be a substitute for both the capacity and archival tier in enterprise databases.
- A CSD-targeted, cache-controlled, multijoin algorithm and associated cache eviction policy that enables adaptive, push-based query execution under out-of-order data arrival at low cache capacities.
- A query-aware, ranking-based I/O scheduling algorithm for CSD that maximizes efficiency and maintains fairness.
- Full system implementation and evaluation of the Skipper framework based on PostgreSQL and OpenStack Swift that shows that Skipper on CSD approximates the performance of a classical query engine when running on the HDD-based capacity tier within 20% on average.

2. BACKGROUND

In this work, we frame our problem in the context of a modern, multitenant, virtualized enterprise data center. In such a scenario, tenants deploy databases in virtual machines (VM) which run on virtualized *compute servers*. Each VM is backed by a virtual hard disk (VHD) that provides storage for both the guest OS image and database files. The VHD itself is stored as a file or a logical volume on a shared storage service that runs on a cluster of *storage servers*. For instance, enterprise data centers use OpenStack's Nova service to deploy VM and Cinder, a scale-out block storage service, for storing their virtual hard disks. Similarly, Amazon hosts database (RDS) VM in EC2 and provides block storage for these VM using Elastic Block Store (EBS).

2.1 Database storage tiering

Most modern enterprise data centers today use a four-tier storage hierarchy (performance, capacity, archival, backup) implemented using three storage types (online, nearline, and offline) as shown in Figure 1. The latency-critical performance tier, typically implemented using DRAM, SSD, or high-performance 15k-RPM SCSI HDD, and the high-density capacity tier, implemented using SATA HDD, are managed by block storage services like Amazon EBS.

In contrast, the archival tier, based on robotic tape libraries (VTL) or optical jukeboxes, and backup tier, based on offline tape cartridges or optical drives, are never used for storing randomly accessed, queryable data. Thus, these devices are managed using a separate service that exposes storage as an object-based blob store. For instance, OpenStack provides Swift, an object-storage service that can be used to store and retrieve objects over a RESTful HTTP interface; Oracle's Storage Tek tape libraries have been extended to expose storage via the Swift interface [30].

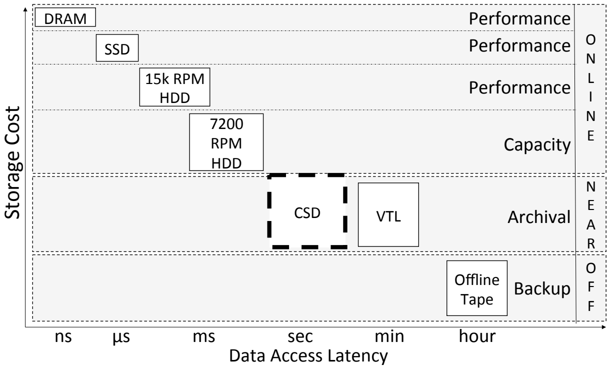

Figure 1: Storage tiering for enterprise databases

The setup described above is typical of modern enterprise data centers and both block and object storage services are shared across several tenants.

Cold data in the archival tier. The proliferation of cold data necessitates low-cost data storage techniques. Given that databases already have a tiered storage infrastructure in place, an obvious low-cost solution to deal with this problem is to store cold data in either the capacity tier or the archival tier. Table 1 and Figure 2 show the cost of building a 100-TB database using various tiered storage strategies as reported by a recent analyst study [42]. The one-tier storage strategy uses only a single storage device for housing all data. The two-tier strategy uses 15k-RPM SCSI disks as the performance tier, and 7,200-RPM SATA disks as the capacity tier with no archival tier. The three-tier strategy spreads data across the two HDD tiers and a tape-based archival tier. Finally, the four-tier strategy uses an SSD to hold the hottest data items in addition to the remaining tiers.

Clearly, any strategy that uses the tape-based archival tier for storing data provides substantial reduction in cost. Storing all data on tape is unsurprisingly the cheapest option that provides a 20× reduction in cost compared to the All-SATA strategy that uses the capacity tier exclusively. Similarly, a disk–tape three-tier strategy that uses the archival tier provides a 2× cost reduction compared to a disk-only two-tier strategy and 1.25× reduction compared to the All-SATA strategy. Note here that savings quickly add up as the database size increases further, motivating the need to store as much cold data as possible in the archival tier.

Application-hardware mismatch. Despite the cost benefits, cold data cannot be stored in the archival due to mismatch between application demands and hardware capability. The archival tier is typically used to store only rarely accessed compliance and backup data. As the expected workload is predominantly sequential writes with rare reads, the high access latency of tape drives is tolerable. Using the archival tier to store cold data, however, changes the application workload, as analytical queries might be issued over cold data to extract insightful results [18]. As a nearline storage device with access latency at least four orders of magnitude larger than the slowest online storage device (HDD), tapes are unable to handle this workload.

Thus, today, databases use the performance tier for storing data accessed by latency-sensitive real-time queries, and the capacity tier for storing data accessed by latency-insensitive batch queries. The archival tier is never used directly during query execution, but only during compliance verification or online media failure.

2.2 Cold storage devices

Over the past few years, storage hardware vendors and researchers have become cognizant of the gap between the HDD-based capacity tier and the tape-based archival tier. This has led to the emer-

	SSD (P)	15k-HDD (P)	7.2k-HDD (C)	Tape (A)
Cost/GB	$75	$13.5	$4.5	$0.2
2-tier	-	35%	65%	-
3-tier	-	15%	32.5%	52.5%
4-tier	2%	13%	32.5%	52.5%

Table 1: Acquisition cost in $/GB and fraction of data stored in each device type for various tiering configurations as reported by [42]. The table also shows the tier corresponding to each device, with *P* standing for performance, *C* for capacity, and *A* for archival

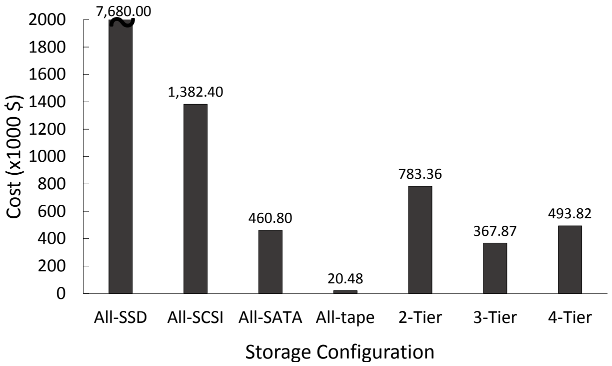

Figure 2: Cost benefits of storage tiering

gence of a new class of nearline storage devices explicitly targeted at cold data workloads. These devices, also referred as *Cold Storage Devices (CSD)*, have three salient properties that distinguish them from the tape-based archival tier.

First, they use archival-grade, high-density, Shingled Magnetic Recording-based HDD as the storage media instead of tapes. Second, they explicitly trade off performance for power consumption by organizing hundreds of disks in a MAID configuration that keeps only a fraction of HDD powered up at any given time [7]. Last, they right-provision hardware resources, like in-rack cooling and power management, to cater to the subset of disks that are spun up, reducing further operational expenses.

Although CSD differ in terms of cost, capacity, and performance characteristics, they are identical from a behavioral stand point–each CSD is a MAID array in which only a small subset of disks is spun up and active at any given time. For instance, Pelican [40] packs 1,152 SMR disks in a 52U rack for a total capacity of 5 PB. However, only 8% of disks are spun up at any given time due to restrictions enforced by in-rack cooling (sufficient to cool only one disk per vertical column of disks) and a power budget (enough power to keep only one disk in each tray of disks spinning). Similarly, each OpenVault Knox [36] CSD server stores 30 SMR HDDs in a 2U chassis of which only one can be spun up to minimize the sensitivity of disks to vibration. The net effect of these limitations is that CSD enforce strict restrictions on how many and which disks can be active simultaneously (referred to as a *disk group*).

All disks within a group can be spun up or down in parallel. Access to data in any of the disks in the currently spun up storage group can be done with latency and bandwidth comparable to that of the traditional capacity tier. For instance, Pelican, OpenVault Knox, and ArticBlue are all capable of saturating a 10-Gb Ethernet link as they provide between 1-2 GB/s of throughput for reading data from spun-up disks [36, 40, 41].

However, accessing data on a disk outside the currently active group requires spinning down active disks and loading the next group by spinning up the new set of disks. We refer to this operation as a *group switch*. For instance, Pelican takes eight seconds to perform the group switch. Thus, the best case access latency of CSD is identical to capacity tier while the worst-case access latency is two orders of magnitude higher.

Given such high access latencies, CSD, similar to tape drives and other nearline storage devices, cannot be used as a primary data store for performance critical workloads. Thus, they have also been integrated in data centers using an object-based blob storage service. For instance, both Spectra's ArcticBlue CSD and Pelican provide a key–value interface for storing GB-sized data blobs [40].

3. THE CASE FOR COLD STORAGE TIER

The price/performance characteristics of CSD raise an interesting question: *How should CSD be integrated into the database tiering hierarchy?* Although an obvious approach involves using CSD as a faster archival tier, enterprise databases could achieve further cost reduction by using CSD to build a new storage tier that subsumes the roles of both the capacity and archival tier. We refer to this new storage tier as the *cold storage tier (CST)*. With such an approach, the three-tier hierarchy that included performance, capacity, and archival tiers would be reduced to a two-tier hierarchy with 15k RPM disks in the performance tier and CSD in the cold-storage tier. Similarly, the four-tier hierarchy would be reduced to a three-tier hierarchy with SSD and 15k RPM disks in the performance tier and CSD in the cold storage tier.

3.1 Price implications of CST

Figure 3 shows the cost reduction achievable by doing this replacement. For performance and capacity tiers the same pricing as listed in Table 1 has been used, while for CSD we use three cost/GB values, namely $0.1/GB (ArcticBlue CSD pricing [25]), $0.2/GB (assuming CSD costs same as tape), and $1/GB (hypothetical worst-case pricing).

As can be seen, the cost reductions are substantial. At $0.1/GB, using a single cold storage tier instead of separate capacity and archival tiers reduces cost by a factor of 1.70×/1.44× for three/four-tier installations. At $0.2/GB, the cold storage tier provides a cost saving of 1.63×/1.40×. Even in the worst case ($1/GB), the cold storage tier provides 1.24×/1.17× cost reduction. In terms of absolute savings, these values translate to hundreds of thousands of dollars for a 100TB database, and tens of millions of dollars for larger PB-sized databases.

3.2 Performance implications of CST

Despite its potential, CSD and the CST they enable will be useful only if databases can run their workloads directly on data stored in CSD. Otherwise, CSD are no better than tape libraries and will be relegated to the role of a fast archival tier. To understand the implications of using a CSD-based CST as a substitute for the capacity tier, one needs to quantify the impact of group switch latency on the database performance and scalability.

Perils of analytics on CSD. In the best case, read requests from the database are always serviced from a HDD that is spun up. In such a case, there would be no performance difference between using a CSD and the traditional capacity tier. However, in the pathological case, every data access request issued by the database would incur a group switch delay and cripple performance.

Unfortunately, the average case is more likely to be similar to the pathological case due to two assumptions made by traditional databases: 1) storage subsystem has exclusive control over data allocation, 2) underlying storage media supports random accesses with uniform access latency. In a virtualized data center that uses CSD as a shared service, both these assumptions are invalidated leading to suboptimal query execution.

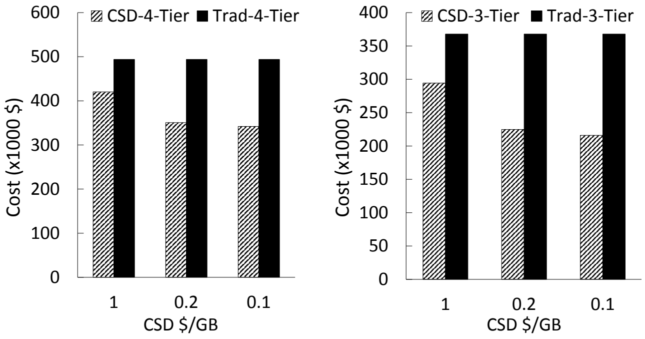

Figure 3: Cost savings of CSD as a replacement for the HDD-based capacity tier

In a virtualized datacenter, a CSD usually stores data corresponding to several hosted databases by virtualizing available storage behind an object interface. Thus, each individual database instance has no control over data layout on the CSD. The CSD might store data pages corresponding to different relations (or even a single relation) in different disk groups due to several reasons. For instance, a CSD might decide to spread out data across different disk groups for load balancing. A set of disks could fail in a group causing the CSD to temporarily stop allocating data in that group until recovery completes, or data could arrive in increments, which could lead to different increments being stored in different disk groups. The lack of control over data layout implies that the latency to access a set of relations depends on the way they are laid out across groups.

Moreover, the CSD services requests from multiple databases simultaneously. Thus, even if all data corresponding to a single database is located in a single group, the execution time of a single query is not guaranteed to be free of group switches. This is due to the fact that the access latency of any database request depends on the currently loaded group, which depends on the set of requests from other databases being serviced at any given time.

Benchmarking CSD. To quantify the overhead of group switches, we setup an experimental testbed that emulates a virtual enterprise data center fully described in Section 5.1. Five servers in our testbed act as compute servers. Each server hosts an independent PostgreSQL database instance (client) running within a VM. We have chosen a one-DBMS-per-VM configuration to isolate performance of each client and avoid any possible resource contention across clients. OpenStack Swift, an object store deployed as a RESTful web service and extended with a custom plug-in, runs on a sixth server acting as our emulated, shared CSD.

For our benchmark, each PostgreSQL instance services TPC-H queries on a 50GB TPC-H dataset. Only the database catalog files are stored in the VM's VHD. The actual binary data is stored in Swift as objects, where each object corresponds to a 1GB data segment, and fetched on demand during execution time. Each client is allotted its own disk group, and all data from a client is stored within its allotted group. Thus, if only one client were using the CSD, it could retrieve objects without any group switches.

Our objective is to measure 1) the performance impact of running many PostgreSQL clients on a shared CSD, and 2) the performance sensitivity to the CSD access latency. To this end, we run two experiments. In both experiments, we use Q12 from the TPC-H benchmark as our workload. This is a two-table join over lineitem and orders, the two largest tables.

For our first experiment, we issue Q12 to all PostgreSQL instances simultaneously (each client with its own data) and measure the observed execution time of each instance. We repeat the experiment 5 times, each time, increasing the number of clients by one. Thus, Swift services GET requests from one PostgreSQL instance during the first run, two instances during the second run, and so on.

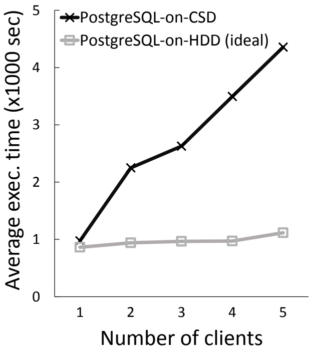

Figure 4: PostgreSQL over CSD vs HDD

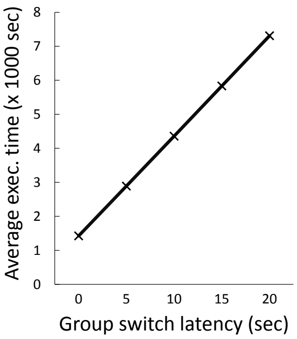

Figure 5: Latency sensitivity

For our second experiment, we fix the number of clients at 5, and perform 5 iterations of Q12, each time varying the group switch time in Swift from 0 to 20 seconds in 5-second increments.

Results. Figure 4 shows the average query execution time as we increase the number of clients with a 10-second group-switch latency when the storage target is either a CSD or a HDD-based capacity tier. The results for the HDD-based case were obtained by configuring the Swift middleware's metadata to map the data of all clients to a single group, thereby eliminating group switches completely. As can be seen, PostgreSQL-on-CSD exhibits poor scalability as the average execution time increases proportional to the number of clients.

This performance drop is due to the fact that each PostgreSQL client requests and processes data one segment at a time in a very specific order. As each client's data is on a different group, Swift processes the GET requests by switching to each group one by one and returning back the segment from that group. Thus, two consecutive requests from any PostgreSQL client are separated by five group switches leading to a significant increase in overall execution time. In fact, if we have C clients, each processing a query involving D data segments stored on a CSD with a group switch time of S, the total execution time of the query would be $S \times C \times D$. Any increase of one of the three parameters results in a proportional increase in execution time. This also explains why PostgreSQL suffers from extremely high sensitivity to the group switch latency as shown by the 6× increase in execution time with a group switch latency of 20 seconds (shown in Figure 5).

Furthermore, the 6× increase in execution time we report is only an optimistic estimate of the performance impact of running queries on CSD. Back-of-the-envelope calculations indicate that PostgreSQL would suffer from a 10-100× increase in execution time compared to the traditional case that employs HDD in the capacity tier, when the CSD group switch latency, number of segments, or number of clients increase. Given such performance implications, it is unclear if the CSD can be used to store even cold data, let alone replace the HDD-based capacity tier. Thus, the only way CSD can be integrated into the enterprise database tiering hierarchy is as a replacement for the archival tier. Unfortunately, such an integration misses out cost-saving opportunities provided by CSD.

3.3 A case for CSD-driven query execution

Clearly, exploiting the cost benefits of CSD while minimizing the associated performance trade off requires eliminating unnecessary group switches. For instance, consider an example layout shown in Table 2, where three relations A, B and C, each containing two objects (data segments), are stored across three groups denoted as $g1$, $g2$ and $g3$ (e.g., $A.2$ denotes an object of relation A stored in group 2). In the optimal case, all three tables can be retrieved from the CSD with just two group switches. However, as the database has no control over data layout or I/O scheduling, the only way of

Group	Table objects
g1	A.1, B.1, C.1
g2	A.2, B.2
g3	C.3

ID	Subplans
1	A.1,B.1,C.1
2	A.1,B.2,C.1
3	A.2,B.1,C.1
4	A.2,B.2,C.1
5	A.1,B.1,C.3
6	A.1,B.2,C.3
7	A.2,B.1,C.3
8	A.2,B.2,C.3

Table 2: Data layout and Execution subplans

achieving such an optimal data access is to have the database issue requests for all necessary data upfront so that the CSD can batch requests and return data in an order that minimizes the number of group switches. Thus, the order in which database receives data, and hence the order in which query execution happens, should be determined by the CSD in order to minimize the performance impact of group switching.

Unfortunately, current databases are not designed to work with such a CSD-driven query execution approach. Traditionally, databases have used a strict optimize-then-execute model to evaluate queries. The database query optimizer uses cost models and statistics gathered to determine the optimal query plan. Once the query plan has been generated, the execution engine then invokes various relational operators strictly based on the generated query plan with no runtime decision making. This results in *pull*-based query execution where the database explicitly requests segments (i.e., pulls) in an order determined by the query plan. For instance, continuing the previous example, PostgreSQL might request all objects of table C first, followed by B, and finally A. This pull-based execution approach is incompatible with the CSD-driven approach, as the optimal order chosen by the CSD for minimizing group switches is different from the ordering specified by the query optimizer. Even more, as pull-based execution is oblivious to data layout, it will invariably cause many more group switches leading to poor performance when CSD is used as the capacity tier. For example, fetching relations C, B, A, in that order leads to 5 switches instead of 2.

4. QUERY PROCESSING ON CSD

Having described the shortcomings of the naive approach to the DB-CSD integration, we now present Skipper, a query execution framework that enables SQL analytics directly on data stored in CSD. As CSD will not be able to service workloads that have strict latency requirements or require fine-grained read/write access to random disk blocks, CSD is an unsuitable storage medium for OLTP installations. Thus, Skipper's target application domain is long running batch analytics with write-once-read-many workloads.

Figure 6 shows the components that constitute the Skipper architecture. As before (Section 3.2), each PostgreSQL instance runs within a VM, is allotted a fixed amount of memory, stores only catalog information in the VHD, and uses the CSD as the storage tier. To address the pull-based execution problem, each database instance now uses a *CSD-driven, cache-aware, multi-way join algorithm (MJoin)* to perform efficient out-of-order query execution. In this paper we focus on join queries, since scans could naturally be serviced in an out-of-order fashion.

The second component, OpenStack Swift, our CSD, is shared across all the tenants, and uses an I/O scheduler that coordinates accesses to data stored in different storage groups. Each database instance tags each request with a *query identifier* to make the CSD workload aware. The CSD uses this information to implement a novel *rank-based, query-aware scheduling algorithm* that balances fairness and efficiency across tenants.

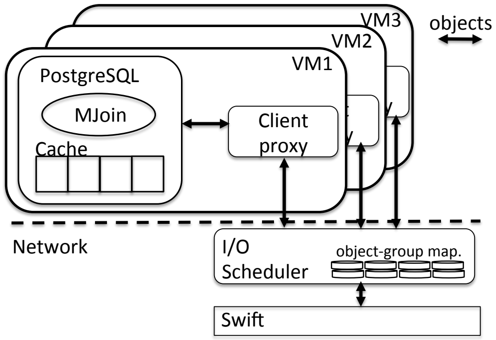

Figure 6: Skipper architecture

In addition to the database and CSD, Skipper introduces a third component, referred to as the *client proxy*. It is a daemon process that is collocated with each PostgreSQL instance in all VM and coordinates communication between MJoin and Swift.

Having described a high-level overview of the Skipper architecture, we now discuss each component in more detail.

4.1 CSD-driven, cache state-aware MJoin

Skipper performs a CSD-driven out-of-order execution of queries by building on recent work done in Adaptive Query Processing (AQP) [9]. AQP techniques were designed to deal with streaming data sources in the Internet domain that pose three issues to the traditional database architecture: 1) unpredictable variations in data access latency, 2) non-repeatable access to data, and 3) lack of statistics or stale statistics about data. In order to overcome these issues, AQP techniques abandon the traditional pull-based, optimize-then-execute model in favor of out-of-order execution [3, 16] and runtime adaptation [2,4,45].

Multiway-Join [45] is one such AQP technique that enables out-of-order execution by using an n-ary join and symmetric hashing to probe tuples as data arrives, instead of using blocking binary joins whose ordering is predetermined by the query optimizer. Under out-of-order data arrival, the incremental nature of symmetric hashing requires MJoin to buffer all input data, as tuples that are yet to arrive could join with any of the already-received tuples. Thus, in the worst case scenario of a query that involves all relations in the dataset, the MJoin buffer cache must be large enough to hold the entire data set. This requirement makes traditional MJoin inappropriate for our use case as having a buffer cache as large as the entire data set defeats the purpose of storing data in the CSD.

Skipper solves this problem by redesigning MJoin to be cache aware. Our *cache-aware MJoin* implementation splits the traditional monolithic MJoin operator into two parts, namely, the *state manager*, and the *join operator*.

Algorithm 1 presents the pseudo-code of the MJoin state manager. At the beginning of execution, state manager retrieves information about all objects (segments) across all tables that are necessary for evaluating a query. This information is typically stored as part of the DBMS' catalog. The state manager uses this information to track query progress by building *subplans*. Subplans are disjoint parts of query execution that can proceed independently and produce query results. For instance, Table 2 shows the set of subplans that would be generated for a query that joins tables A, B and C, each of which has two segments. Each combination of each relation segment makes a subplan. The state manager creates all such subplans and tags them as pending execution.

After generating subplans, the state manager issues requests for all objects needed for executing the query and waits to receive *any* of the requested objects. Upon the arrival of an object, the state manager checks to see if enough cache capacity is available to buffer the new object. If so, the state manager builds appropri-

```
Algorithm 1: MJoin: State manager algorithm
Input: $Q$: query, $cache\_size$: cache size
Output: $R$: result tuples
// Initialization
$cache$ = alloc($cache\_size$)
$issue\_queue$ = readObjectsFromCatalog($Q$)
$pending\_spl$ = makeSubplans($issue\_queue$)
while $issue\_queue \neq NULL$ and $pending\_spl \neq NULL$ do
    SwiftGetObjects($issue\_queue$)
    $issue\_queue$ = NULL
    while $swift\_obj$ = ReceiveObjectFromSwift() do
        if inPending($swift\_obj, pending\_spl$) then
            if cacheIsFull($cache, swift\_obj$) then
                $dropped$=cacheEvict($cache, swift\_obj$)
                if inPending($dropped, pending\_spl$) then
                    $issue\_queue$ =addToQueue($dropped$)
            addObjectToCache($swift\_obj$)
            $runnable$ =readySubplans($cache, pending\_spl$)
            if $runnable \neq NULL$ then
                $R$+=joinExecute($runnable$)
                movePendingToExecuted($runnable$)
return $R$
```

ate hash tables based on the join conditions and projection clauses in the query, and populates the hashtable using tuples from the new object. If the cache is full, the state manager uses the cache eviction algorithm described in Section 4.2 to pick a target object and frees space by dropping its hashtable. If the evicted object takes part in any pending subplan, a request for the object reissue is appended to the queue of requests that will be reissued in the next cycle (i.e., upon completing all pending requests issued previously).

Next, the state manager checks if there are any subplans that can proceed with execution based on the current cache state. Subplans are moved into runnable state when all objects comprising them are present in the database cache. Should that be the case, those subplans are executed by triggering join execution. The n-ary join operator in itself is stateless. The state manager instructs the join operator to probe the set of hashtables corresponding to the objects being joined. Once the actual join operation is completed, the subplan moves into *executed* state.

This process repeats until all requested objects have been received. At that point, if there are pending subplans left, the state manager reissues requests only for those objects necessary to execute the pending subplans and continues execution until no further subplans are left.

4.2 Cache management

When operating under limited cache capacity, MJoin might need to evict previously fetched objects in order to accommodate new arrivals. If such an evicted object is needed by a pending subplan, it will be refetched again from the CSD. As repeated refetching can deteriorate performance, we need a cache replacement algorithm that minimizes the number of reissues. It is well-known that the offline problem of determining an optimal order for fetching/evicting disk pages for performing a two-table join, given limited main memory, with the goal of minimizing the number of disk I/O, is NP-complete [27]. In our case, the order in which data arrives is controlled by the CSD, and can change dynamically depending on data layout and concurrent requests from other clients. Thus, designing an optimal online eviction algorithm is impossible given that the order of data arrival is non-deterministic even across two different executions of the same query. In designing our caching algorithm, we opted for greedy heuristics that could exploit the fact that the MJoin state manager has full visibility of both cache contents and pending subplans.

Maximal number of pending subplans. Our first algorithm was based on the intuition that it is beneficial to prioritize objects that participate in a large number of pending subplans over less popular ones. We illustrate this policy with an example. Consider the example configuration shown in Table 2. Let us assume that we have (A.1, B.1, A.2, C.3) stored in our cache of capacity 4 and we have already processed subplans <A.1, B.1, C.3> and <A.2, B.1, C.3>. When the next object arrives, we need to decide if it should be cached, and if so, pick an eviction candidate for replacement.

Assuming $C.1$ arrives next, if we count the total number of pending subplans per object, we get 4 for $C.1$, 3 for $A.1$ and $A.2$, and 2 for each $B.1$ and $C.3$. Thus the algorithm would consider $B.1$ and C.3 as viable eviction candidates. If the algorithm picks $C.3$ as the eviction candidate, $C.1$ would be accepted and MJoin can make progress by executing new subplans. However, should the algorithm pick $B.1$ for eviction, MJoin would have been unable to proceed with any of the subplans as there would be no objects belonging to table B.

Maximal progress. Our preliminary results highlighted this problem and provided us the insight that evicting objects just based on the total number of pending subplans is impractical especially at low cache capacities. Thus, we designed a new progress-based cache management algorithm that picks as eviction target the object that participates in the *least number of executable subplans* given the current cache state and the newly arriving object.

Continuing the previous example, the number of executable subplans given the cache state (A.1, B.1, A.2, C.3) and the new object $C.1$ is 1 for each $A.1$ and $A.2$, and 2 for $B.1$, as they would trigger the execution of subplans < $A.1$, B.1, C.1> and <A.2, B.1, C.1>, but 0 for $C.3$. Thus, this policy would pick C.3 as the eviction candidate since it has the lowest number of executable plans. If the algorithm finds more than one object with the same number of executable plans, it uses the number of pending subplans to break ties. A beneficial side effect of our maximal progress algorithm is that it automatically prioritizes small tables over large ones as objects belonging to the small table participate in many more subplans. As typical data warehousing workloads follow star or snow flake schema, where a large central fact table is joined with many small dimension tables, this caching policy would automatically reduce the number of reissues by keeping small tables pinned in the cache.

4.3 Client proxy

The client proxy is a mediator between MJoin and CSD. When MJoin maps each relation to a list of objects that need to be fetched from Swift, it serializes the list of object names into a JSON string, passes the list over a shared message queue to the client proxy, which, in turn, submits HTTP GET requests to fetch these objects from Swift. In this way, the client proxy offers an interface-independent mechanism for connecting PostgreSQL with a CSD.

Furthermore, the client proxy shares semantic information with Swift, i.e., it generates a query identifier for each set of requests from PostgreSQL and tags each Swift GET request with this identifier. This enables the scheduler to identify all objects being requested as a part of a single query, which allows it to implement semantically-smart scheduling (as described in Section 4.4).

Once MJoin submits requests to the client proxy, it blocks until it is notified of data availability. As each GET request completes, the client proxy notifies MJoin of the availability of a data object. Although we could have modified the scan operator or the storage backend of PostgreSQL to communicate with Swift, we chose the MJoin-client proxy route for an additional reason. By blocking the execution at the MJoin operator, we make the whole out-of-order execution mechanism data-format and scan-type independent. For instance, while we use binary data files and default PostgreSQL scan operators for the purpose of this paper, we have used the same framework to query Swift-resident, raw data files directly using the scan operator provided by File Foreign Data Wrapper in PostgreSQL without changing any Skipper component.

4.4 Scheduling disk group switches

At any given point in time, the CSD receives a number of requests for different objects from various database clients. As these objects could potentially be spread across different storage groups, the CSD has to make three decisions: 1) *which group* should be the target of the next group switch?, 2) *when* should a group switch be performed?, and 3) *what ordering* should be used for returning objects within a currently loaded group? When choosing a proper scheduling strategy, our goal is to identify a scheduling algorithm that balances efficiency and fairness in answering these questions.

Which group to switch to? The CSD group scheduling problem can be reduced to the single-head tape scheduling problem in traditional tertiary storage systems, where it has been shown that an algorithm that picks the tape with the largest number of pending requests as the target to be loaded next performs within 2% of the theoretically optimal algorithm [35]. If efficiency was our only goal, we could use an algorithm that always chooses a disk group housing data for the maximum number of pending requests as the target group to switch. We refer to it as the *Max-Queries* algorithm.

However, we need a mechanism to provide fairness, in the absence of which, a continuous stream of requests for a few popular storage groups can starve out requests for less-popular ones under the Max-Queries algorithm. Current CSD solve this problem by scheduling object requests in a First-Come-First-Served (FCFS) order to provide fairness with some parameterized slack that occasionally violates the strict FCFS ordering by reordering and grouping requests on the same disk group to improve performance [40]. Although such an approach might be sufficient when dealing with archival/backup workloads, it fails to provide optimal performance for our use case, since a single query requests many objects, which a query-agnostic CSD treats like independent requests. Thus, enforcing FCFS at the level of objects would produce many unwarranted group switches in an attempt to enforce fairness and prevent request reordering optimizations we describe later in this section.

As mentioned earlier, the Skipper client proxy tags each GET request with a query identifier making the Skipper scheduler workload aware as it knows which object requests correspond to which queries. Thus, one option to provide fairness would be to use a query-based FCFS algorithm rather than an object-based one. Such an algorithm, however, would be inefficient as it fails to exploit request merging across queries (servicing all requests in a group before switching to the next one), and produces many more group switches than Max-Queries (as shown later).

Rank-based, query-aware scheduling. Our new scheduling algorithm strikes a balance between the query-centric FCFS algorithm and the group-centric Max-Queries algorithm by integrating fairness into the group switching logic. In our new scheduling algorithm, each group is associated with a rank R and the scheduler always picks the group with the highest rank as the target of a group switch. The rank of a group g, denoted as *R(g)*, is given as:

$$R(g) = N_g + K(\sum_{q=1}^{N_g} W_q(g)) \tag{1}$$

where N_g is the number of queries having data on group g, K is a constant whose value we derive shortly, and $W_q(g)$ is the waiting time of a query that has data on group g, defined as the number of group switches since the query was last serviced. Thus, any query which is serviced by the current group will have 0 waiting time.

In order to understand the intuition behind this algorithm, let us consider the two parts of the equation separately. The first part, N_g, when used alone to determine the rank gives us the Max-Queries algorithm we described earlier. The second part provides fairness by increasing the rank of groups that have data belonging to queries which have not been serviced recently. Each time the scheduler switches to a new group g, a set of queries S_g become serviceable, and the remaining queries non-serviceable. As the non-serviceable queries have to wait for one (or more) group switches, their waiting time increases. By directly using their waiting time to determine the rank, the algorithm ensures that groups whose queries have long waiting times have a higher probability of being scheduled next.

The scaling factor K determines a tipping point between efficiency and fairness and we will now derive its value. Consider two sets of queries Q_1 and Q_2 requesting data on groups g_1 and g_2 respectively such that set Q_2 arrives s group switches after set Q_1. Let t be time of arrival of Q_2 and let R(g_1) and R(g_2) be the rank of the two groups at t. If the scheduler follows a strict FCFS policy, it would schedule Q_1 before Q_2 at time t irrespective of the number of requests to each group (N_{g_1}, N_{g_2}). Thus, if R(g_1) was greater than R(g_2), the scheduler's behavior would be similar to the FCFS policy. This naturally leads to the following implications:

$$\implies N_{g_1} + K * W_{g_1} > N_{g_2} + K * W_{g_2} \,\text{where}\, W_{g_i} = (\sum_{q=1}^{N_{g_i}} W_q(g_i))$$
$$\implies K > (N_{g_2} - N_{g_1})/(W_{g_1} - W_{g_2})$$
$$\implies K > (N_{g_2} - N_{g_1})/s \,\text{as}\, Q_2 \text{ arrives } s \text{ switches after } Q_1$$

Thus, we need to pick a value of K in the $\left[0, (N_{g_2} - N_{g_1})/s\right]$ range, to balance fairness and efficiency. To maximize efficiency, the scheduler should switch to group g_2 at time t for all $N_{g_2} > N_{g_1}$. Thus, the scheduler should ensure that the following holds:

$$R(g_2) > R(g_1) \forall N_{g_2}, N_{g_1} \,\text{where}\, N_{g_2} > N_{g_1}$$
$$\implies N_{g_2} + K * W_{g_2} > N_{g_1} + K * W_{g_1}$$
$$\implies K < (N_{g_2} - N_{g_1})/(W_{g_1} - W_{g_2})$$
$$\implies K < (N_{g_2} - N_{g_1})/s \forall N_{g_2}, N_{g_1} \text{where} N_{g_2} > N_{g_1}$$

Thus, if we choose K < 1 / s, we are guaranteed that the scheduler will switch to group g_2 and service the set Q_2 when $N_{g_2} > N_{g_1}$. Recall that s is the number of group switches between the arrival of sets Q_1 and Q_2. As $s \to \infty$, K → 0, and the algorithm tends to favor efficiency over fairness as the rank degenerates to N_g. For the minimum value of s = 1, which translates to K = 1, the algorithm's fairness is maximized. Therefore, we set the value of K to 1.

When to switch? Given a set of active requests for objects in a currently loaded group, the scheduler has to decide whether to service all requests or a partial subset before switching to the next group. We favor the approach of avoiding preemption, since our problem is similar to the tertiary I/O scheduling problem, where it has been shown that preemption leads to suboptimal scheduling [35]. Thus, once we switch to a group, we satisfy all object requests on that group before switching to the next group.

What ordering within a group? Although MJoin can handle out-of-order data delivery, the order in which objects are returned plays an important role in determining execution time. For instance, consider a query over three tables A, B and C, where each table has three objects, all of which are stored in the same group. Let us assume that the database can cache only three objects. If the scheduler first returns all objects of A, then all objects of B, and finally all objects of C, the MJoin implementation will be forced to reissue requests for several objects repeatedly even with our efficient cache management algorithm, as the MJoin cannot make progress with objects belonging to the same table. On the other hand, if the I/O scheduler returned back objects in a semantically-smart fashion, satisfying object requests evenly across all the relations (e.g. $A.1, B.1, C.1$, then $A.2, B.2, C.2$, etc), the number of reissues would be much smaller as the MJoin implementation can execute many subplans. Thus, our scheduler implements *semantically-smart* ordering within a loaded storage group.

5. EXPERIMENTAL EVALUATION

We now present a detailed experimental analysis of the Skipper framework to prove the effectiveness of various algorithms used in Skipper by comparing its performance with vanilla PostgreSQL.

5.1 Experimental Setup

Hardware. In all our experiments, we used five servers equipped with two six-core Intel Xeon X5660 Processors, @2.8 GHz, with 48GB RAM, and two 300GB 15000-RPM SAS disks setup in a RAID-0 configuration as our compute servers.

Our shared CSD service is hosted on a DELL PowerEdge R720 server running RHEL6.5, equipped with two eight-core Intel E5-2640 processors clocked at 2GHz, 250GB of DDR3 DRAM, and a hardware RAID-0 array of seven 250GB SATA disks with a peak throughput of 1.2GB/s. All servers are connected by a 10GB switch.

Software. Each compute server runs Ubuntu 12.04.1 and hosts a virtual machine that is allocated four processing cores and a 100GB VHD. The VM runs Ubuntu 14.04.2 LTS cloud image as the guest OS and PostgreSQL 9.2.1 as the database system. There are two versions of PostgreSQL installed in each VM, one extended to support MJoin and the other being the default one. Henceforth, we will refer to MJoin-enabled version of PostgreSQL as Skipper. We limit the amount of memory allocated to the VM to 1GB over the cache size allocated to PostgreSQL to ensure that: 1) the guest OS has enough buffer space to prevent swapping and 2) our MJoin code works as expected with a limited amount of memory. Vanilla PostgreSQL is always configured to use "effective-cache-size" of 30-GB, "shared buffers" and "'work memory" of 16GB.

Compute Servers. Only the database catalog files are stored in each of the VM's VHD. The actual binary data stored in a set of files, one set per relation, where each file in the set represents a 1GB segment (default Postgres segment size) of the relation, is stored in Swift as objects and fetched on demand during execution time. Each relation has a corresponding Swift container, and each segment is stored as an object within the container. Containers/objects are named based on the filenode identifiers used internally by PostgreSQL to map relations and their segments to files.

In order to connect PostgreSQL with Swift, we wrote a FUSE file system that intercepts local file accesses from PostgreSQL and translates them into Swift HTTP-GET calls. For instance, when PostgreSQL scans through a relation, it accesses the backing files one segment at a time. On receiving the first read call for a segment, the FUSE file system uses the segment's file name (same as filenode number) to derive the container/object names and issues a HTTP-GET call to fetch the corresponding object from Swift.

Shared CSD. We built an emulated CSD by extending Swift using a Python middleware that provides MAID-like functionality. We used OpenStack Swift v2.4.1 in the Single-All-In-One Swift setup to get all Swift processes (Proxy, Account, Container, Object servers) running in our CSD server. The middleware groups disks into disk groups based on a configuration file and maintains persistent metadata to map each object to its disk group. If the middleware receives a GET request for an object on the currently active group, it services it immediately by forwarding it to the Swift backend. However, if it gets a request for an object in a different group, it emulates a group switch by artificially adding delays to the request processing path instead of actually spinning up/down disk drives. In addition to maintaining disk-to-disk group mapping and object-to-disk group assignment metadata, the middleware plugin also implements the I/O scheduling algorithms (see Section 4.4).

Benchmarks. We used four benchmarks, namely, TPC-H [43] with SF-50, Star-Schema Benchmark (SSB) [28] with SF-50, a popular data analytics benchmark [31] over 20GB database, and a genome-sequencing benchmark over a 13GB NREF database [48].

5.2 Experimental Results

We present the results in the following order. First, we show the benefit of out-of-order execution by comparing Skipper to vanilla PostgreSQL. Then, we present a sensitivity analysis of Skipper's algorithms to the group switch latency, layout, cache and data set size. Last, we show a comparative evaluation of our scheduling algorithm to show the benefit of using the rank-based scheduling.

5.2.1 Benefit of out-of-order execution

Figure 7 shows the average query execution time of Skipper on CSD, PostgreSQL on CSD (marked 'PostgreSQL') and PostgreSQL on HDD configurations (marked 'Ideal') under TPC-H Q12. Similar to Figure 4, we scale the number of clients from 1 to 5. We configured both PostgreSQL and Skipper implementation to use a cache size of 30GB (half the dataset size), and the Swift scheduler to use a one-group-per-client data layout, where all data corresponding to a single client is stored together in one group, while data from different clients lies in different groups. As can be seen, Skipper scales much better than PostgreSQL as we increase the number of clients. At five clients, Skipper outperforms PostgreSQL by a factor of 3 when CSD is used as the storage backend. In addition, Skipper is only 35% slower than the ideal HDD-based configuration despite the 10-second CSD group switch time.

Figure 8 shows the cumulative execution time of a batch of queries under a mixed workload. For this experiment, each client runs a different workload (TPC-H Q12, JoinTask from the analytical benchmark, Q1 from SSB, and a 4-table join that counts protein sequences matching a specific criteria from NREF) repeating the workload query 5 times. The results are similar to Figure 7, as Skipper provides from 2-3× reduction in execution time in all cases.

The scalability of Skipper in these results can be attributed entirely to the ability of MJoin to perform out-of-order query execution. Under both PostgreSQL and Skipper, Swift switches to each group one by one and services the HTTP GET requests. But in contrast to the vanilla PostgreSQL, our MJoin-enabled PostgreSQL in Skipper can handle out-of-order data arrival. Thus, it submits requests for all necessary data blocks upfront to Swift enabling Swift to service GET requests for all objects within a group before switching to the next group. As a result, the total waiting time for any client C is $(C-1) \times (D/B+S)$, where D is the total number of objects in the dataset and B is the rate at which Swift can push objects out to the client, and S is the group switch latency. Vanilla PostgreSQL, on the other hand, would have a total execution time of $C \times S \times D$, as we explained in Section 3.2.

Figure 9 shows the average execution time breakdown per client for the 5 clients case, each client running TPC-H Q12 (as in Figure 7). 'Switch time' and 'Transfer time' both constitute the waiting time of each DB instance to receive its data, while 'Processing' goes to actually processing the query. As it can be seen, 98% of the total execution time in the case of PostgreSQL is spent on waiting, out of which 65% is the CSD switch time. On the contrary, Skipper optimizes the switch time, reducing it to a mere 2%, while 41% of the total execution time of Skipper goes to useful work. In the case

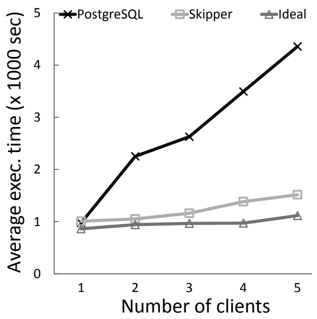

Figure 7: Average exec. time comparison

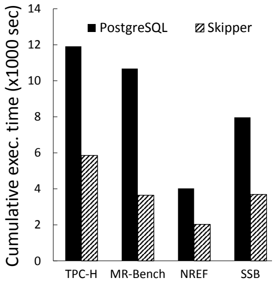

Figure 8: Cumulative exec. time of mixed workload

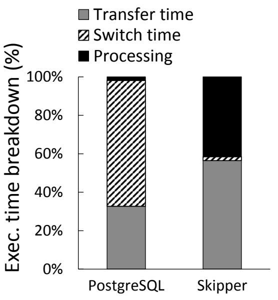

Figure 9: Avg. exec. time breakdown for 5 clients

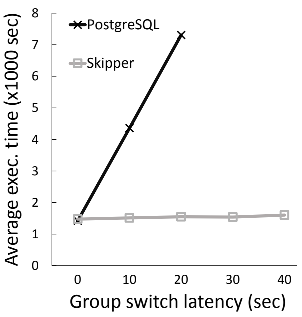

Figure 10: Sensitivity to CSD group switch latency

of Skipper, the biggest stall actually goes to receiving data from Swift (shown as 'Transfer time').

To understand the overhead of each component present in the system we run three experiments with Q12 TPC-H: 1) all data is stored locally and accessed directly using the native file system, 2) all data is stored locally but accessed using our FUSE file system (this applies to vanilla PostgreSQL as MJoin does not use the FUSE file system), and 3) all data is stored remotely on Swift within a single disk group (i.e., there will not be any group switches). We use the results from the experiments to break down execution time of PostgreSQL and Skipper into 1) query execution, 2) FUSE file system, 3) network access as shown in Table 3.

Comparing query execution times under PostgreSQL and Skipper, we see that in the absence of group switches, Skipper takes 25 seconds more than PostgreSQL which translates to 6% overhead, showing that out-of-order query execution in Skipper has marginal overhead. FUSE file system itself adds very little overhead (1.6%) to PostgreSQL's execution. Last, storing data remotely in Swift doubles the execution time under both PostgreSQL and Skipper.

Our current Swift middleware explicitly serializes GET requests and services them one at a time to simplify implementation and ensure correctness of emulation (for instance, ensuring that the Swift backend has no pending requests for the current group before emulating a group switch). As a result, it does not overlap disk I/O with network I/O and substantially increases the end-to-end transfer latency. We verified this by running PostgreSQL on default Swift without the Skipper middleware and we found that the execution time was only 25% higher than the local run. However, as the overhead induced by the middleware is common to both PostgreSQL and Skipper, reducing it would provide proportional improvement in execution time in both systems. Thus, the relative performance of the two systems and insights we derive in the paper will not change under an optimized plugin implementation.

There are two important conclusions we would like to draw here. First, based on the above equations, it is clear that if $D/B >> S$, Skipper will make the database clients insensitive to access latency. As we target analytics over large data sets stored in CSD, this will be the common case. PostgreSQL, on the other hand, will always suffer for even minor increase in C, S or D. Second, we are neither saturating the storage I/O throughput (1.2GB/s) nor the network bandwidth (10Gb/s) with our current Swift middleware. Thus, by parallelizing the servicing of requests within a group, we can reduce transfer time substantially. With such improvements, Skipper would outperform PostgreSQL by a big margin and offer performance comparable to conventional disk-based storage services.

5.2.2 Sensitivity to the group switch latency

Figure 10 shows the average execution time of Skipper and PostgreSQL with five clients, under TPC-H Q12, as we increase the group switch latency from 10 to 40 seconds. Comparing Figure 10 and Figure 5, one can see that Skipper is tolerant to the changes in group switch latency. These results validate our previous claim that Skipper makes database clients insensitive to access latency. This, again, is due to the fact that the I/O scheduler is able to minimize the number of group switches by serving all requests in a single group before switching to the next group. Thus, unlike the PostgreSQL case, where there were a total of 57 group switches (one per segment accessed), the MJoin case has only five group switches. Because of its tolerance, Skipper can even work with CSD with much higher group switch latencies.

Component	PostgreSQL		Skipper	
Query execution	407s	41.9%	433s	43%
Fuse file system	15.75s	1.6%	/	/
Network access	550s	56.5%	574s	57%

Table 3: Execution breakdown of PostgreSQL and MJoin

5.2.3 Sensitivity to the layout choice

Figure 11a shows the average query execution time of both systems as we vary the layout in the CSD. We obtain these results by fixing the number of clients to 4 and varying the layout in the Swift scheduler. We use 4 layouts for these experiments, namely, *all-on-one* (Allin1), *two-clients-per-group* (2perG), *one-client-per-group* (1perG), and *incremental* (Increm.). The first three layouts gradually expand the clients out across one, two, and four groups. The last layout partitions each client's data into two parts and stores each half on separate groups such that group *G1* stores *C1.1* and *C4.2*, *G2* stores *C1.2* and *C2.1*, *G3* stores *C2.2* and *C3.1*, and *G4* stores *C3.2* and *C4.1*.

There are two important observations to be made. First, notice that both Skipper and default PostgreSQL have similar execution time under the all-in-one case as there are no group switches. However, in all other cases, Skipper provides 2× to 3× improvement over vanilla PostgreSQL. Second, the execution time under PostgreSQL increases progressively as we unroll the data across groups from the all-in-one case to the one-client-per-group case. This shows the impact that layout has on default PostgreSQL. Under Skipper, execution time increases between the all-in-one case and two-clients-per-group case due to data transfer delays as we mentioned earlier. However, it remains constant as we fan out from two to one client per group, proving the low sensitivity of Skipper to variations in layout.

5.2.4 Sensitivity to the cache size

We now present results quantifying the impact of cache size on our MJoin implementation. For this experiment, we fix the number of clients to five, configure Swift to use the one-client-per-group layout, and use TPC-H Q5 as our benchmark. We choose Q5, since it is a complex six-table join whose input size almost covers the

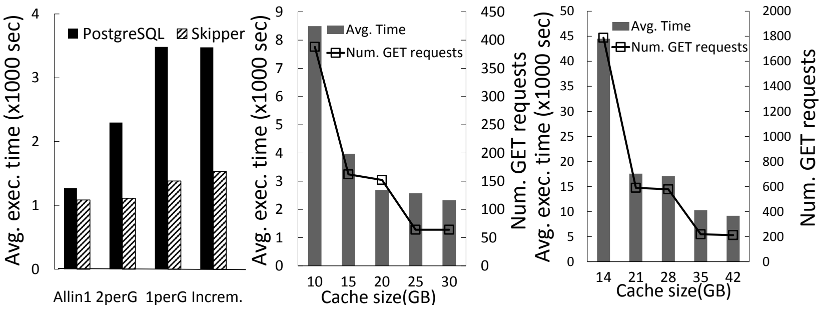

Figure 11: MJoin sensitivity to: a) layout b) cache size c) data set size

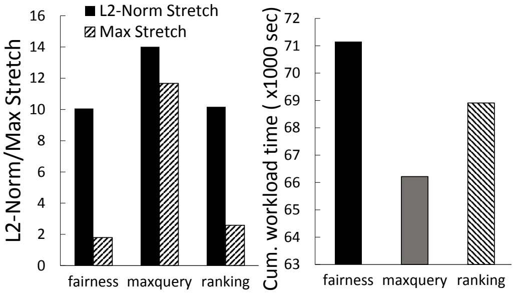

Figure 12: Fairness vs. efficiency: a) L2-Norm b) Cumulative workload time

whole TPC-H data set and produces many more subplan combinations compared to the Q12.

Figure 11b shows the average execution time of MJoin at various cache sizes. The average query execution time under vanilla PostgreSQL was 3,710 seconds (not shown). Thus, in the worst case (10GB), Skipper is 2.2× slower than PostgreSQL. It matches PostgreSQL's performance at 15GB (20% of data set) and provides a 1.37× to 1.59× improvement at higher cache sizes.

To test the scalability of MJoin at low cache capacities, we repeat the same experiment with a larger data set (TPC-H SF-100). As before, we run Q5 that reads 127 objects out of 140 in total, varying the cache size from 14 objects (10% of data set size) to 42 objects (30% of data set size) in 5% increments. There are 14630 subplans in total. The results are presented in Figure 11c.

Comparing Figure 11b and Figure 11c, we can see that Skipper's execution time increases substantially as we reduce the cache size. Under SF-50, Skipper's execution time increases 3.6× as we shrink the cache size from 30GB(40%) to 10GB(14%). Under SF-100, Skipper's execution time increases 4.8× as we scale down the cache size further from 42GB(30%) to 14GB(10%). The performance drop is a consequence of the increased number of request reissues as shown by the black line in Figure 11b and Figure 11c. Under SF-50, the total number of Swift objects requested by MJoin increases from 64 to 388 as we reduce the cache size. SF-100 pushes this further as MJoin requests 212 objects at 42GB and 1787 objects at 14GB cache capacities respectively.

These graphs show an important trade-off between the cache capacity and performance of MJoin. Given R relations each of size S objects, the traditional hash join has a time complexity of $O(S \times R)$ as each relation is fetched only once and used to either build or probe a hash table at each hash join stage. This requires cache capacity (C) to be large enough to hold all (but one) relations, i.e., it requires a cache capacity of $S \times (R-1)$. In the best case, MJoin is able to buffer $R-1$ input relations in memory entirely and avoid request reissues completely. Thus, similar to hash join, its best case time complexity is $O(S \times R)$ for the cache capacity of $S \times (R-1)$. However, unlike hash join, MJoin can proceed even with limited cache capacities at the expense of performance.

Let us consider a cache of capacity C $<< (R-1) \times S$. Given the small cache capacity, MJoin will proceed in several cycles. In each cycle, MJoin will request all objects belonging to pending subplans and execute the subplans as objects arrive. Let us consider one such cycle. As our query-aware scheduler returns data corresponding to relations in a round-robin fashion, the cache is evenly divided among R relations, with $\frac{C}{R}$ objects of each relation being buffered. To simplify the analysis, let us assume that join execution happens after C objects have arrived in the cache. Thus, we get the first batch of C objects. Given $\frac{C}{R}$ objects of R tables in the cache, $(\frac{C}{R})^R$ subplans are evaluated. Then, we get the next batch of C objects and perform join execution. Given the relation size of S, this process repeats $\frac{S \times R}{C}$ times in each cycle. Thus, a total of $(\frac{C}{R})^R \times (\frac{S \times R}{C})$ subplans will be evaluated in each cycle. Given that the total number of subplans is S^R, the number of cycles (reissues) that will happen is then $\frac{S^R}{(\frac{C}{R})^R \times (\frac{S \times R}{C})} = (\frac{R \times S}{C})^{(R-1)}$. MJoin needs C to be large enough to hold at least R objects so that at least one subplan can make progress. Thus, in the worst case, with a cache capacity of R, the time complexity of MJoin is $O(S^R)$. Comparing this with the best case $(O(S \times R))$, we can see the trade off between the cache capacity and performance of MJoin.

Despite the fact that request reissue can be high, there are techniques to decrease its overhead. In the case of TPC-H queries we tested, the request reissue was high as each table object contains tuples contributing to the end result. Thus, the same object is refetched and rescanned multiple times. Should the distribution of result tuples differ in a way that interesting tuples are clustered rather than being uniformly distributed across all objects, Skipper would substantially reduce the request reissue overhead. In such a case, Skipper marks the objects not containing any result tuples and omits requests for this object in the future, pruning out subplans in which it takes part. For instance, let us consider a 4-table join with 10 objects in each table. The total number of possible subplans is 10^4. Nonetheless, if even a single object does not have data that contributes to the result, Skipper can safely prune 10^3 subplans (as all subplans with that object are guaranteed not to produce any result). This subplan pruning, combined with the fact that Skipper automatically prioritizes caching of small tables over large ones, will ensure that the performance drop due to reissues is not dramatic even in the case of big data sets. In the case of TPC-H queries we tested such subplan pruning did however not occur. Thus, request reissue dominated execution.

5.2.5 Balancing efficiency and fairness

Our final result shows the effectiveness of our ranking-based scheduling algorithm in balancing fairness and efficiency. For this experiment, we use 5 clients, each issuing TPC-H Q12 ten times. We configure Swift to use a skewed data layout such that two groups have data corresponding to two clients each, and the last group stores the fifth client's data. We compare three scheduling algorithms, namely, FCFS ('fairness'), Max-Queries ('maxquery'), and our Rank-based algorithm ('ranking'), all of which were explained in Section 4.

In addition to reporting query execution time, we also report L2-norm [6] stretch as a performance metric for comparing alternative algorithms. L2-norm stretch is defined as follows:

The L2-norm of stretch for a workload consisting of queries q_i : i = 1...n with stretches $s_i : i = 1...n$ is equal to $\sqrt{\sum_{i=1}^{n} s_i^2}$

In scheduling theory, *stretch* of a job is defined as the ratio of the observed execution time to the ideal execution time, where the ideal execution time is the time taken to execute the job *alone* on the platform. Stretch essentially shows the deviation from the optimal case due to negative impact of interaction between jobs. L2-

norm then aggregates stretch values across several clients enabling us to use a single metric for comparing the pros and cons of different experimental configurations (considering both the average and maximum values of stretch). In our case, the ideal execution time of a query is the single-client execution time, as all requests from a single PostgreSQL instance can be serviced by the CSD without any group switches. The stretch for cases with more than one client is obtained by normalizing the observed execution time by the single-client execution time.

Figure 12a shows both L2-norm and maximum stretch across the three scheduling policies, while Figure 12b shows the cumulative execution time across all clients. As expected, the 'Max-Queries' algorithm has the lowest execution time but significantly increases maximum stretch, as queries on the group with just one client end up starving. The 'FCFS' algorithm, in contrast, trades off efficiency for fairness as evidenced by the reduction in maximum stretch but proportionate increase in overall completion time. Our rank-based scheduling algorithm adopts a middle ground. Initially, the algorithm sticks to the two groups with two pending queries, thus, maximizing efficiency. However, each time Skipper switches to one of these two groups, it also increases the rank of the group with just a single client by one. Once every four group switches, the group with a single client outranks the rest, resulting in the corresponding client being serviced. Thus, the 'Rank-based scheduling' algorithm balances efficiency while avoiding starvation.

Summary. From all the experiments, we can clearly see that the Skipper architecture scales better and tolerates higher group switch latencies than the traditional architecture when CSD is used as a primary storage. All three aspects of Skipper (out-of-order execution, efficient caching, and rank-based scheduling) substantially contribute toward masking the CSD group switch latency.

6. RELATED WORK

Integration of DBMS and CSD considered in this paper naturally draws an inspiration from a large body of work coming from the database and the storage systems world. In the following we discuss avenues mostly related to the architecture of Skipper.

Tertiary databases. Integration of databases and CSD mostly resembles the work on tertiary memory databases [37, 38]. However, unlike tapes that are exclusively accessed and controlled by a database system, CSD is shared among multiple tenants, making a tight pull-based control between caching and scheduling impossible. Similarly, while in tertiary databases the notion of tenant-fairness is nonexistent, in the enterprise data centers fairness should be of a primary concern.

Adaptive query processing. Adaptive query processing emerged in the past decade as a way to deal with environmental changes, either as a way to fix suboptimal query optimization decisions taken a priori during compilation procedure [5, 20, 22, 26], or as a way to deal with the changes in data arrival characteristics often appearing in streaming environments [2, 4, 15, 44, 45, 47]. Neither of the techniques is, however, fully applicable to the DBMS-CSD integration. While the former approaches address the orthogonal issue of cardinality misestimates in query optimization, the latter trade off adaptivity for high memory consumption. To operate under limited cache capacity, the execution strategy, however, has to be tightly coupled with the cache management. In this paper, we thus propose a cache-controlled MJoin algorithm that efficiently supports out-of-order data arrival even at low cache capacities.

Scheduling theory. The problem of scheduling could be considered at several levels of granularity: starting from the low level block-based or I/O scheduling to tape-based and finally job or task scheduling at higher granularity levels.

Considering the first, proportional share schedulers [21, 39] allocate throughput (IOPS) to each application in proportion to user-specified weights. Further efforts extend it with reservation and limits to provide flexible bounds on resource allocation for virtualized storage [13] and with techniques for balancing fairness and throughput [12]. In contrast, [34, 46] use time multiplexing instead of fair queuing to provide strict performance isolation under interference from multiple workloads. All these approaches assume that I/O requests are directed at a set of disks that are spun up. Our work, in contrast, focuses on an orthogonal problem of scheduling object requests at a higher level, i.e., across both spun up and powered down disks with the goal of minimizing the total number of spin ups while balancing fairness. Thus, our scheduling algorithm can potentially be extended with these complementary approaches to perform proportional sharing within a disk group.

Tape scheduling algorithms have so far focused only on reducing the number of switches while ignoring fairness. Recent research has shown that an optimal algorithm for scheduling a given set of I/O requests over a set of tapes is the one that minimizes the number of switches, and that an algorithm that picks the next device as the one with the largest number of pending requests constantly performs within 2% of the optimal algorithm [35]. We adopt this algorithm, i.e., the policy that chooses to service next the group with the maximal number of queries, and extend it with a notion of fairness. Our ranking based algorithm was inspired by the rFEED [14] task scheduling algorithm. The problem of task scheduling has been studied extensively in the past [33]. However, while task scheduling algorithms assume that task execution times are independent, query execution time in our case depends on which group is active at the time of query execution, which, in turn, depends on the order of query execution. Thus, task scheduling algorithms are not directly applicable to our context.

Hot and cold data classification and migration. There is a large body of research involving data classification in the context of main-memory databases or multitiered databases that can be used to identify *hot* and *cold* data, e.g. [8, 10, 24]. Enterprise databases have long used such algorithms to improve performance by caching hot data in low-latency storage devices. Similarly, database have also used Hierarchical Storage Managers (HSM) to automatically manage migration of data between online, nearline, and offline storage tiers [23]. We do not consider the orthogonal problems of data classification or automatic data migration in this paper. Rather, we focus on query execution over "cold data at rest" in the CSD after classification and migration has taken place.

7. OUTLOOK AND CONCLUSIONS

In this paper, we demonstrate how the usage of cold storage devices enables a new tier in the enterprise storage tiering hierarchy, named the *cold storage tier*. We show that the cold storage tier is able to replace both the capacity and archival tiers in their functionality, thereby offering major cost savings for enterprise data centers. Furthermore, we show that data analytics can be run on such a platform by a judicious hardware-software codesign where both the database query execution engine and the CSD work in concert toward achieving a common goal – masking the high access latency of CSD group switches.

The implications and benefits of using CSD reach far beyond enterprise data centers, and are equally applicable to cloud providers. For instance, Cloud Service Providers (CSP) have already started deploying custom-built, rack-scale CSD explicitly targeted at cold data workloads [11, 32, 36]. By doing so, CSP have already reported substantial cost savings. For instance, according to a recent report from Facebook, the Open Vault cold storage system reduced their expenses by a third compared to conventional online storage;

their Blu Ray-based cold storage system reduced power consumption by 80% over Open Vault [32]. Recognizing the potential of CSD, CSP have started offering hosted, low-cost cold storage services based on CSD, and such *cold-storage-as-a-service* offerings are quickly gaining popularity, offering cloud customers a chance to benefit from inexpensive storage [1, 11].

We believe that the CSD benefit for cloud providers could go beyond offering just storage-as-a-service. By following the design and architecture of Skipper presented in this paper, CSP could offer cloud-hosted *data analytics services over CSD*. Such a design would benefit both customers and providers of cloud-hosted data analytics services, as providers could increase revenue by offering cheap analytics services on data stored on CSD, and customers could reduce total cost of ownership by running latency insensitive analytics workloads on cold data stored on CSD.

ACKNOWLEDGMENTS

We would like to thank Prof. Donald Kossmann, the anonymous reviewers, and the DIAS laboratory members for their constructive feedback that substantially improved the presentation of the paper. We thank Javier Alejandro Rivas for his help with MJoin implementation. This work is partially funded by the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement number 617508 (ViDa) and the European Union Seventh Framework Programme (FP7 Collaborative project), under grant agreement number 317858 (BigFoot).

8. REFERENCES

[1] Amazon. Amazon glacier. http://aws.amazon.com/glacier/.

[2] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *DIS*, 1996.

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: A data-centric analytic processing engine for large data warehouses. In *SIGMOD*, 2010.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.

[6] N. Bansal and K. Pruhs. Server Scheduling in the Lp Norm: A Rising Tide Lifts All Boat. In *STOC*, 2003.

[7] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Conference on Supercomputing*, 2002.

[8] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[9] A. Deshpande, I. Zachary, and V. Raman. Adaptive Query Processing. In *Foundations and Trends in Databases*, 2007.

[10] A. Eldawy, J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. In *VLDB*, 2014.

[11] Google. Google cloud storage nearline. White paper, 2015.

[12] A. Gulati, A. Merchant, M. Uysal, P. Padala, and P. Varman. Workload dependent io scheduling for fairness and efficiency in shared storage systems. In *HiPC*, 2012.

[13] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, 2010.

[14] C. Gupta, A. Mehta, S. Wang, and U. Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *EDBT*, 2009.

[15] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, 1999.

[16] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[17] IDC. Technology Assessment: Cold Storage Is Hot Again Finding the Frost Point, 2013.

[18] IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014.

[19] Intel. Cold Storage in the Cloud: Trends, Challenges, and Solutions. White Paper.

[20] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, 2004.

[21] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS*, 2004.

[22] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[23] B. Laliberte. Automate and optimize a tiered storage environment fast! ESG White Paper.

[24] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[25] S. Logic. Arctic blue pricing calculator. https://www.spectralogic.com/arcticblue-pricing-calculator/.

[26] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing through Progressive Optimization. In *SIGMOD*, 2004.

[27] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *VLDB*, 1981.

[28] P. O. Neil, B. O. Neil, and X. Chen. Star Schema Benchmark. 2009.

[29] S. Newsletter. Costs as barrier to realizing value big data can deliver. http://www.storagenewsletter.com/rubriques/market-reportsresearch/37-of-cios-storing-between-500tb-and-1pb-storiantresearch-now/.

[30] Oracle. Openstack swift interface for oracle hierarchical storage manager. White Paper, 2015.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[32] PCWorld. Facebook uses 10,000 blu-ray discs to store 'cold' data. http://www.pcworld.com/article/2092420/facebook-puts-10000-bluray-discs-in-lowpower-storage-system.html.

[33] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[34] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, 2008.

[35] S. Prabhakar, D. Agrawal, and A. El Abbadi. Optimal scheduling algorithms for tertiary storage. *Distributed and Parallel Databases*, 14(3):255–282, 2003.

[36] O. C. Project. Cold storage hardware v0.5, 2013.

[37] S. Sarawagi. Query Processing in Tertiary Memory Databases. In *VLDB*, 1995.

[38] S. Sarawagi and M. Stonebraker. Reordering Query Execution in Tertiary Memory Databases. In *VLDB*, 1996.

[39] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems*. *Real-Time Systems*, 22(1):9–48, 2002.

[40] R. B. Shobana Balakrishnan, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *OSDI*, 2014.

[41] Spectra. Arcticblue deep storage disk. Product, https://www.spectralogic.com/products/arcticblue/.

[42] H. I. Strategies. Tiered storage takes center stage. Report, 2015.

[43] TPC. Tpc-h benchmark. http://www.tpc.org/tpch/.

[44] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[45] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, 2003.

[46] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[47] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. In *PDIS*, 1991.

[48] C. H. Wu and et al. The Protein Information Resource: an integrated public resource of functional annotation of proteins. *Nucleic Acids Research*, 2002.