Secondary Indexing Techniques for Key-Value Stores: Two Rings To Rule Them All

Joseph Vinish D'silva joseph.dsilva@mail.mcgill.ca
Roger Ruiz-Carrillo roger.ruiz@mail.mcgill.ca
Cong Yu cong.yu@mail.mcgill.ca
Muhammad Yousuf Ahmad muhammad.ahmad2@mail.mcgill.ca
Bettina Kemme kemme@cs.mcgill.ca
School of Computer Science, McGill University
Montréal, Canada

ABSTRACT

Secondary indices are traditionally used in DBMS to increase the performance of queries that do not rely on the keys of the table for data reads. Many of the newer NoSQL distributed data stores, even if they provide a table-based data model such as HBase, however, do not yet have a secondary indexing feature built in. In this paper, we explore the challenges associated with indexing modern distributed table-based data stores and investigate two secondary index approaches which we have integrated within HBase. Our detailed analysis and experimental results prove the benefits of both the approaches. Further, we demonstrate that such secondary index implementation decisions cannot be made in isolation of the data distribution and that different indexing approaches can cater to different needs.

CCS Concepts

•**Information systems** → **Point lookups; Unidimensional range search;**

Keywords

secondary indexing; NoSQL data stores; in-memory indices;

1. INTRODUCTION

Secondary indexing plays a key-role in addressing the performance necessities of relational database management systems (RDBMS). It is instrumental in facilitating efficient selection of a subset of the dataset based on business constraints by providing alternative access paths to the base records. Its performance benefits are primarily derived from the reduced I/O facilitated by retrieving only those data pages that contain relevant records.

The dawn of BigData resulted in a resurgence of interest in NoSQL (non-relational) DBMS, in particular, *key-value* stores. Their simple data model and query interface allowed for easy distribution of data, and thus, scalability. Also other typical DBMS functionality, such as transaction management, was only implemented in rudimentary format, leading to much less complexity compared to traditional RDMBS. However, given the prolific success and decades of domination of the relational model, applications started to request the modeling and querying functionality they were used to from RDBMS but at the same time wanted to maintain the flexibility and scalability of *key-value* stores.

Therefore, several "hybrid" data stores emerged that adapted a relaxed notion of the "table-column" abstraction (we will cover this further in section 2.2) with a much less restrictive table-based data model than traditional RDBMS, suitable for the sparse datasets that are commonly found in BigData applications. In tandem with this table concept the query interface also became more powerful, allowing, e.g., for predicate search. However, answering such complex queries either requires scanning the entire data sets or access to alternate access paths. As a result, many NoSQL stores are trying to implement their own notion of a secondary index.

A notable deviation in the development of key-value stores compared to the traditional DBMS is its open source nature, facilitating the database development community at large to pitch in their contributions. This has resulted in various design approaches being attempted in providing a secondary index functionality for these data stores, including HBase [14] (which we briefly cover in related work in section 6). However, so far, we are not aware of any in-depth study and comparative analysis of these different approaches. Moreover, many implementations fall short on modularity. Our analysis of index implementations of HBase shows that most of them require significant changes to the core HBase code instead of leveraging the existing HBase frameworks to develop a pluggable module.

In this paper, we present an in-depth discussion of indexing for distributed, table-based NoSQL data stores. In particular, our paper makes the following contributions.

- We discuss two indexing strategies for distributed key-value stores: one based on distributed tables that is able to exploit the table model of the underlying system for index management, the other using a co-location approach allowing for efficient main-memory access.
- Both strategies are implemented and integrated into HBase in a non-intrusive way.

- We provide an enhanced client interface to query HBase tables using secondary indexing that supports both point queries and range queries.
- We present a detailed performance metrics on various database operations with secondary indices and a comparative analysis of the different approaches.
- We present a thorough analysis on the effects of data distribution on different indexing approaches.

2. BACKGROUND

2.1 Indexing Techniques

Predicate queries that only retrieve a subset of the data of a table can be executed through a table scan where each record is inspected and only qualifying records are returned. If the table is already sorted (clustered) based on the attribute on which the search constraint is defined, then a full table scan can be avoided as the matching tuples can be found in logarithmic time.

Alternatively to a scan, special index structures can help identify the records that qualify, and then only those records are retrieved from the base table. Indices are defined over one or more attributes. Such indices are called *secondary indices* and are often constructed over non primary-key attributes. Typically, a secondary index contains an entry for each existing value of the attribute to be indexed. This entry can be seen as a key/value pair with the attribute value as key and as value a list of pointers to all records in the base table that have this value. In centralized database management systems (DBMS) a pointer is typically a physical identifier indicating the position of the record in the file system. In distributed systems or more high-level implementations, a pointer is often the primary key of the record, assuming that a lookup via the primary key can be done efficiently.

A simple index implementation would be a system table (inverted list) with the index attribute as primary key and the list of record pointers as extra column. This system table can be sorted by primary key making the lookup of a specific attribute value fast. More common are tree-based indices, where inner nodes guide to leaf nodes which contain the key/value pairs. Both sorted inverted lists and tree structures are good for point queries (search on a single attribute value) and range queries (search on a range of attribute values). A mechanism that performs very well for point queries is a hash-table with the index attribute as key and the list of pointers as value.

In some cases, a query can be answered directly using the index, without having to access the base table. Typically, index structures are effective if few records qualify the search criteria. Through the index, these records are found very quickly and then can be individually retrieved from the base table. When many records qualify, it might be faster to scan the entire table, as a scan allows reading tuples in batches (e.g., all on a given block) while using an index not only first requires the index access but the individual retrievals of the qualifying tuples might result in many random accesses across the blocks of the base table. In fact, if every block contains a qualifying record, an index will not reduce the I/O costs for bringing blocks into main memory.

2.2 Table-based NoSQL DBMS

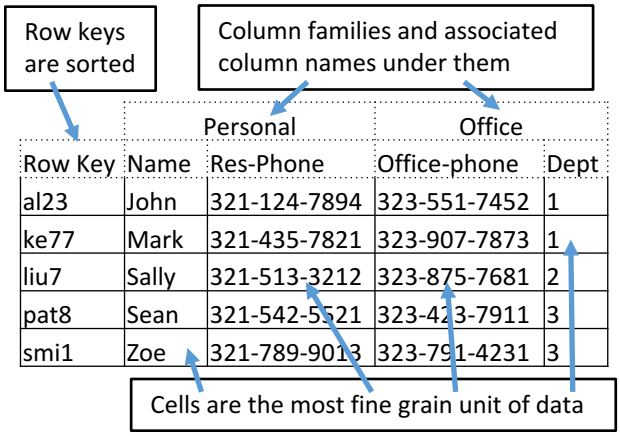

Figure 1: Key-value Data Model [16]

Although NoSQL DBMSes have been around since the 1960s [18] employing hierarchical, network, graph and other semi-structured data models, the dawn of Big Data in the recent decades and the associated processing frameworks have spawned a new breed of NoSQL DBMS, *key-value*, and derived from there, table-based data stores. The resurgence of interest in NoSQL DBMSes was primarily attributed to the rigid structural requirements of relational DBMS, such as the need to define the layout of the data in advance. Such restrictions did not fit well with the processing needs of many modern Big Data applications, prompting the search for alternative data models. Further, scalability requirements necessitated the distribution of data and computation. Widely used in practice are now data stores with a flexible table-based data model. Examples are Bigtable [11], Cassandra [17] and HBase. In the following, we describe the general structure along HBase as the others are quite similar.

2.2.1 *Data Model*

Similar to the relational database data model, HBase aggregates related data into tables. Each table is comprised of multiple rows. Each row contains a unique *row key* (or primary key), one or more column families and columns under them. The table is (logically) sorted by row key. In general, each specific value in the columns is defined as a cell, and can be referenced uniquely by the combination of row key, column family, and column.

An important characteristic of the data model is that not all columns need to be provided for a row, and new columns can be added on the fly. Column names are stored along with the associated value in the database. Such tables can conceptually be sparse as a given column need not be present in many of the rows. Also, column values are treated as byte arrays and clients are responsible to perform the proper data type conversion. This dynamic structure is a distinguishing characteristic compared to RDBMS where the table structure is rigid, typically defined ahead and maintained as meta-table information.

Fig. 1 shows an HBase table containing employee information, with two column families: `Personal` with columns `Name` and `Res-Phone`, and column family `Office` with columns `Office-phone` and `Dept`.

2.2.2 *Client API*

While not providing a full SQL interface, HBase and other table-based NoSQL DBS provide reasonably sophisticated

call level interfaces. The most common data retrieval is the *lookup* method that requires the row key as input. HBase has internal index structures that very efficiently find a record based on its row key. Further, the relational equivalent of *projection* is accomplished by providing the lookup method with the list of column families and columns of interest.

Apart from lookup by primary key, HBase also provides a *scan* operator that can take as input filters similar to SQL predicates. HBase then returns all records that satisfy the filter. Queries over several tables, however, are not provided. A scan operation on a range of row keys is relatively efficient as HBase sorts a table by primary key. However, filters over non row key attributes require a scan over the entire table to find the matching records. There are first attempts of providing secondary indices but none has been officially integrated. Details are discussed in section 6.

Writes and Updates are the same from the perspective of the client API. They require as input the row key, the column families and column names to be updated and their corresponding values. Deletes require the row key of the record impacted, and also the column family names/column names to be deleted. This allows specific attributes to be deleted instead of the entire record [14], a necessity when the data model allows the table to be sparse. Deletes in general do not translate to physical deletes. Instead, a delete marker is used to indicate that the data was requested to be removed. Retrieval operations will encounter the delete marker and skip that record. Storage optimization operations are often performed in a periodical manner which take care of physically removing the deleted data.

2.2.3 Distributed Architecture

In order to handle very large tables, most data store implementations split the rows of a table into multiple *shards*, called *regions* in HBase. Each region can then be served by one of the nodes in the DBS cluster, referred to as *region server* in HBase, and each region server can host several regions, also from different tables. In principle, this is similar to the horizontal partitioning method employed by large scale relational DBMS.

The decision of which shard a particular row belongs to is often determined by a *partitioning function* over the row key. The two common partitioning approaches are based on *hashing* of the row key or on the *range* of the row keys. The later is an especially popular approach when locality of associated rows are desirable (such as retrieving over a small range of row keys). Such requests can often be processed by searching just one shard. HBase follows this approach. When a region becomes very large, the region server can initiate a *region split*, where the rows are divided into two equal sized daughter regions each covering a sub-range of the rows. One of the regions will then typically be reassigned to a different region server.

HBase also performs vertical partitioning, as each column family is stored in a different partition, referred to as *store*. Such vertical partitioning approaches have been known to provide better performance [8] when reading only a subset of columns, a concept popularized by columnar DBMS [7]. To utilize this approach, column families are chosen so as to bundle columns that are frequently accessed together.

There are two fundamental approaches to cluster design. In the first approach, a dedicated node acts as a master. This is the case with Bigtable and HBase. The master server manages meta-information, is responsible for the allocation of regions to region servers and balances the load in the HBase cluster. Clients only communicate with the master to retrieve the meta information (e.g. the location of regions) but they perform data transfer directly with region servers so that the master does not become overloaded. Master failures are handled by a monitoring system external to the DBMS cluster.

In the decentralized approach, there is no master node to perform explicit coordination, but each node in the cluster has identical functionality and can take up the role of a coordinator. Clients connect to any node in the cluster. Apache Cassandra follows this approach of cluster management.

2.2.4 Storage Structure

Many table-based NoSQL data stores follow a storage approach first proposed in the Log-Structured Merge-Tree (LSM-Tree) [22]. In this approach, the table abstraction has an in-memory component and a persistent storage component. The in-memory component is used to store the most recent updates of data. Additionally, these updates are also recorded in the database logs for durability, similar to traditional DBMS. When the in-memory component becomes very large, this data structure is persisted in an immutable format into the disk. Therefore, at any time a table might be made up of one in-memory component and multiple immutable disk components. A search for a row key is first performed on the memory component followed by the disk components. A background process can periodically scan and merge multiple disk components of a table into one larger data structure in the disk to reduce the amount of structures to be searched during a read operation.

In HBase terminology, the in-memory component is called *memstore*, and the persistence storage is in Hadoop file system (HDFS) [9]. HDFS is a fault tolerant distributed filesystem that is designed to run on commodity hardware and is optimized for large datasets. It forms an integral part of the Hadoop ecosystem, providing storage functionality for many distributed applications such as HBase. When the memstore starts filling up, the data is flushed to HDFS where it is stored as immutable HFiles. Thus, a store is a collection of memstore and the HFiles of the corresponding column family. The background process merging these HFiles is referred to as compaction.

3. INDEXING APPROACHES

Secondary indexing becomes complicated in a distributed DBMS. As discussed in section 2.1, a straightforward mechanism stores the inverted list in a separate table and treats it as a "system table" automatically maintained by the index maintenance modules. System table and base table are treated as independent units by the DBMS and thus, in a distributed setting, can reside on different nodes. A second approach *co-locates* index and base table in such a way that the index entries for a record are guaranteed to reside on the same node as the base record itself. In the following sections we will discuss the relevance and the general design approach to both these forms of secondary indexing along with their pros and cons.

3.1 Table-based Secondary Indexing

In the table-based approach, the secondary index can be viewed as special system table whose row key is the sec-

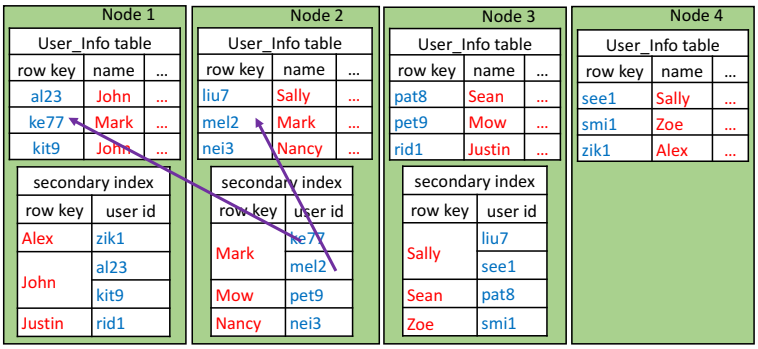

Figure 2: Table-based Secondary Index

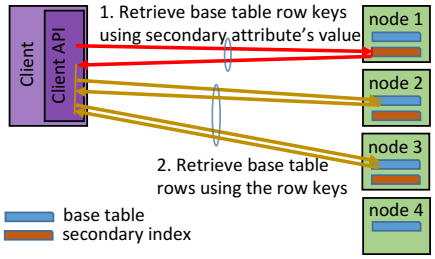

Figure 3: Querying Using a Table-based Secondary Index

ondary attribute's value and an extra column contains the list of row keys of the base table records that contain this secondary attribute's value. By using the DBMS' table management module also for index tables, they can be partitioned and distributed across the cluster in the same manner as base tables. Fig. 2 shows a concrete example with a base table `User_Info` that is partitioned across 4 nodes, and a secondary index for attribute `name`, which is distributed across three of the nodes. Notice how the secondary index entry for `Mark`, which is stored in node 2, points to two base table records, one in node 1 and another in node 2.

Fig. 3 shows the control flow for a read request that utilizes such a table-based secondary index. Read requests based on a secondary attribute's value are sent directly to the node responsible for the secondary index entry corresponding to that value. This node can then either return the keys of the base table records to the client which can then lookup each of the base table records by key (as shown in fig. 3) from the corresponding nodes, or the node itself can read the base table records and send the results back to the client. As can be inferred, this is a *four-hop* process that leads to four consecutive message exchanges (the retrieval of the base records from different nodes can be done in parallel).

An important feature is that base table or index partitions with no relevant data are never involved. The index lookup is only sent to the index partitions that maintain attribute values that are requested: for a point query (e.g., `name = 'Mark'`) this will be one partition, for a range query (e.g. `name` LIKE `'M%'`), the index entries might span more than one partition. After that, only the nodes that have base partitions with matching records are contacted.

If the number of matching base table records is fairly small compared to the total number of partitions, this is an efficient approach as it avoids communication messages to nodes that have no data to return. However, if the base records to be returned span most of the partitions, there is little gain in terms of message exchange compared to a table scan. This is particularly true, as we have the additional round of index access, that might itself contact many partitions in case of large range queries.

Whenever a record is inserted into the base table or an indexed attribute of a record is updated, the corresponding secondary index (indices) must be updated. These updates involve non-negligible communication overhead, as in a large cluster, most secondary index entries will be located in a different node than its base table record. This can lead to contention in index updates, as with non-unique and skewed data distributions, the probability of multiple concurrent updates on the same secondary index entry will rise, making the node handling that index entry a hot spot.

A big advantage of this approach is that it is easy to implement as it can exploit the table management mechanisms already provided by the DBMS.

3.2 Co-located Secondary Indexing

A co-located index adheres to the *shared nothing* architecture [24], a design paradigm that is followed by most modern distributed DBMS, owing to its overall performance and scalability benefits. In this approach, the secondary index entries are stored on the same node as the corresponding base table records. Each node is therefore responsible for maintaining its portion of the secondary index. Fig. 4 shows a co-located secondary index for the same base table as in fig. 2. Notice how the secondary index has two entries for the value `Mark`, one in node 1, the other in node 2.

The control flow for a read request using a co-located secondary index is shown in fig. 5. A read request has to be multicast to all the nodes in the system that contain at least one partition of the base table as any partition can contain a matching record, making it necessary for all the partitions to search their co-located portion of secondary index entries. If a node finds the value in its portion of secondary index, it will lookup the corresponding records in the local base table partition and return them to the client. As the secondary index search and base table retrieval steps are executed locally, a read request in this design has *two-hops*, as the individual searches on partitions can be executed all in parallel. This is, in principle, better than the four-hop cost of table-based indexing. However, if a table has many partitions across many nodes, the message and index lookup costs on all index partitions can be very high. If at the end only few records are returned, the benefits of co-location in terms of message rounds might not outweigh the additional message and processing costs.

A general advantage of this approach compared to a table-based index is that the writes to the base table are less expensive, as the secondary index entries are updated locally, without the communication overhead between the nodes.

A potential disadvantage is that it cannot reuse the table management module of the underlying DBMS but has to be implemented from scratch. Nevertheless, this allows plenty of opportunity for optimization.

3.3 Choosing The right approach

Table 1 shows a summary of the various costs associated with both index types in a cluster of p partitions.

From the table and our previous discussion, we can derive that the table-based approach will be beneficial when there

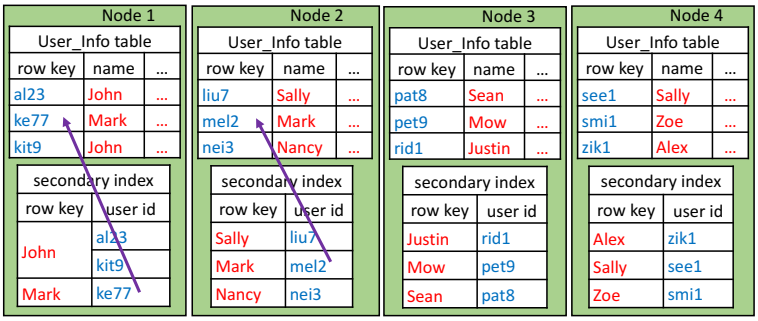

Figure 4: Co-located Secondary Index

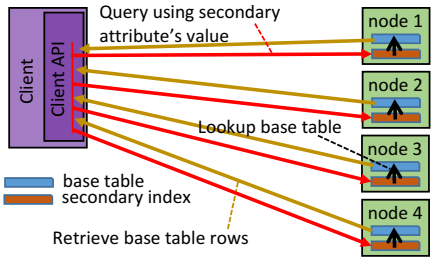

Figure 5: Querying Using a co-located Secondary Index

		Nodes to be contacted	
Type	Hops	unique base row	base rows $> p$
table-based	4	2 (1 index / 1 base)	$1^{+} + p$
co-located	2	p	p

Table 1: Index type overheads for read requests

user_info_personal_name_idx	
row key	i
	p
Mark	{ ke77, mel2 }
Sally	{ liu7, see1 }

Figure 6: Structure of table-based index

are many partitions and queries only return few records as this will lead to few messages and low processing costs as only a few relevant partitions are targeted. In contrast, co-location will be beneficial when there are generally few partitions or queries return records from most partitions.

We can also consider some advanced use cases for secondary indices, such as the ability to use two indices simultaneously to satisfy conjunctive queries. This can be performed easily in the case of co-located indices as the index rows for both the indices will be in the same node (as the node responsible for the base table partition). However, in the case of the table-based approach, the DBMS will have to bring the various index rows of a base table record from different nodes together to perform this operation.

Teradata, a very successful commercial parallel RDBMS, follows the co-located secondary indexing approach for attributes with non-unique values, while using a strategy similar to the table-based approach for attributes that have unique values [25].

4. SECONDARY INDEXING FOR HBASE

We integrated both indexing approaches into the HBase data store. We used HBase tables (HTable) for the table-based index, and we were able to reuse most of the functionality already provided by HTable. For the collocation approach, we implemented our own solution as significant changes to the underlying HTable distribution would have been necessary to exploit it for collocation, and we did not want to change the HBase source code.

Among the HBase recommendations to tackle indexing needs [5], only the co-processor framework [20] option can keep track of updates to base table near real-time. The co-processor framework allows code to be injected into the system without changing the base code, akin to triggers, stored procedures and aspect-oriented programming. It is therefore suited for developing a modular and pluggable indexing solution. We also extend the HBase client interface to allow for the creation of indices and index-based queries.

4.1 Table-based Secondary Index

A table-based index for a secondary attribute is implemented via an HTable. Each distinct attribute value is represented as one row with the attribute value as row key. The index table has one column family `i` with one column `p`. The value stored under `p` is a set of row keys of the base table records, in the serialized form of a java TreeSet [10]. The TreeSet is based on the Red-Black tree data structure, and costs only $\theta(\log n)$ for inserts, updates and deletes, where as a list-based structure would facilitate an $O(1)$ insert but costs $\theta(n)$ to perform updates and deletes [12].

Fig. 6 shows two example rows for an index created on the `name` column of the `personal` column family of the `user_info` table. The row keys of the content table are the userids of the user information stored in the table. There are two rows in the index table. The first row's key '`Mark`' is a value of the `name` column from the `user_info` table, and the set of two userids associated with it are the row keys of the records in the `user_info` table with name '`Mark`'.

It must be noted that, in this approach, a write in the base table can also result in a write in the index HTable, thereby increasing the I/O of the overall write operation.

4.1.1 *Querying Using the HTable Index*

In our implementation, it is the client who decides whether a query should use an existing index or whether HBase should perform a standard table scan. For that purpose we extended the HTable interface of the HBase client API. We have created a new query method that has the same input parameters as the default query (table name and filter predicates that select specific rows), plus an additional parameter that indicates the column name that is indexed.

Query execution is then controlled through the client. For instance, if the filter predicate is a point query (e.g., `name = 'Mark'`), the HBase client library first sends a lookup request to the HTable containing the secondary index, to find the secondary entry with the requested attribute value as row key. This query returns the serialized TreeSet of the base table row keys associated with this secondary index value. The set of returned row keys is then used by the client library to perform a batched lookup for all matching rows on the base table. If the base table has many regions, one batched lookup is sent to each region that contains at least one matching row. The client library then collects all results and returns them to the client application.

Support for range queries is implemented in a similar fashion. In this case the filter contains the start and end range of

the secondary attribute to be constrained. The client library first translates this to a range query on the secondary index with row keys in the search range. This query might be sent to one or several regions of the HTable index. Each region will return the qualifying TreeSets. Once the row keys of all matching records are determined, the procedure is the same as for a point query where the batched lookups are sent to all relevant partitions of the base table, and the results are assembled before return to the user.

Note that if a query contains predicates for indexed attributes and predicates for attributes which have no indices, additional processing is required. There are many ways to achieve this. We first determine the row keys of rows that match the attributes that are indexed. We then push the remaining filters down to the HBase server by the means of modified batch lookups. We are in the process of optimizing this rudimentary approach.

4.1.2 *Region Management*

The split or merge of base table regions or index table regions has no impact on index management, as the approach only works with row keys, and HBase automatically redirects lookups to the appropriate regions. The split of a table region is independent of the split for an index region as for HBase, these are two different HTables. Thus, region management is completely transparent and orthogonal to index management.

In general, HBase decides automatically when to split regions and on which nodes to put them. Thus, in principle, the index designer does not need to be concerned at all with region management. However, HBase allows the user to specify multiple regions at HTable creation time and this might be useful when creating indices in order to evenly distributed the load across region servers, and to prevent a small number of region servers becoming a hot spot for index lookups.

4.2 Co-located In-Memory Secondary Index

In the co-location approach, each base table region (partition) has a collocated index that has index entries covering exactly the rows in the base table region.

Although it is possible to create an HTable with exactly this index content, HBase does not provide any straightforward means to enforce that this HTable-based index partition would be collocated with the corresponding base table region. This is because, by default, HBase considers these as independent HTables and decides individually on their location. We would need to modify the HBase code to implement a custom load balancer, which we did not consider an attractive solution. Instead, we implemented our propriety, optimized index structure and embedded it transparently into the HBase system using the co-processor framework.

Our index structure has been motivated by various aspects. First, we decided to focus on main-memory indexing. That is, the entire index partition is kept in main memory and only persisted during a region shutdown. As a result, index maintenance is very fast and can exploit optimized main-memory data structures. Our motivation is that we are looking at applications that use HBase's capability to scale to many nodes in order to keep the working data set in main memory and avoid expensive I/O. In such settings it should be feasible to keep index structures in memory, too. Thus, our co-located index is memory resident and is persistent only during a region shutdown.

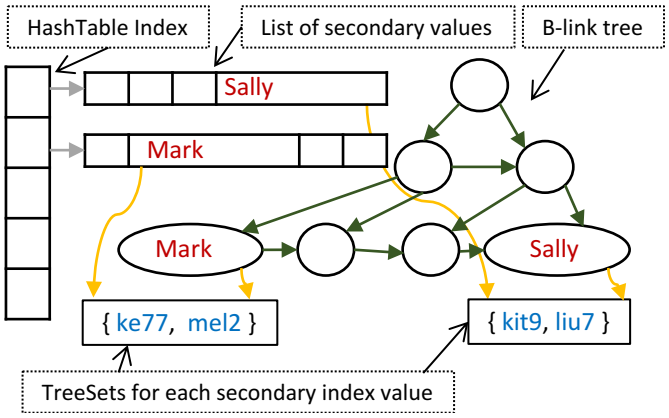

Figure 7: Abstract Layout of the Co-located In-Memory Secondary Index

Second, we wanted to support both point and range queries, and make updates very fast. While hash table based data structures are capable of doing point lookups in $\theta(1)$, they cannot be used for range queries. B^+-trees, on the other hand, are capable of performing range queries efficiently as they store the secondary index values in a logically sorted order. But point queries have a cost of $\theta(n \log n)$. Therefore, we create a hybrid data structure that is a combination of a hash table and a variation of B^+-tree to provide the best of both worlds. As before, the keys to these data structures are the secondary index values themselves. We store the set of base table row keys associated with a secondary index value using a TreeSet, exactly as we did in the table-based design. Each base table region has its own secondary index partition that references only the base table rows which it serves. Our tree-index structure is based on B^{link}-tree, a variation of B^+-tree proposed in [19] and stated to have the highest concurrency and overall optimal performance according to [15, 23]. For the sake of brevity, we encourage the avid reader to refer to their original work for more details.

Fig. 7 shows an example data of the in-memory index data structure. The set of row keys associated with each secondary index value is stored in a TreeSet. The references to the TreeSets are stored in the hash table as well as the B^{link}-tree. Point queries can be easily satisfied by hash table lookups. The hash table also provides faster access to TreeSets for updates and deletes. However, B^{link}-tree can facilitate range queries based on the secondary index value.

4.2.1 *Querying Using the Co-located Index*

The client API for using the co-located index is similar to that of the table-based index. The internal execution flow, however, is more similar to the one for native HBase table scans than for table-based indexing. When the client API receives a query that wants to use a co-located index, it sends a special request containing filter conditions contained in the query to all the regions of the base table. These special requests are sent in parallel so that the execution across all regions can execute concurrently[1].

At the server side, the special request is directed to a co-processor method that performs a lookup over the hash-

[1]Note that the native HBase implementation sends the query to one region after the other not allowing for simultaneous scans across all regions

table portion or a range query over B$^+$-tree portion of the memory resident secondary index structure associated with the base table region to retrieve the base table row keys. These row keys are used to fetch the base table rows from the corresponding region locally. Queries that contain complex filters that span attributes with index and attributes without index, can be easily handled with co-located indices. The indices are used to determine the row keys of rows that fulfill the search criteria on the indexed attributes. Then, the base table is accessed to retrieve those rows and return only those that fulfill the remaining criteria on the other attributes.

4.2.2 *Persistence, Recovery and Region Splits*

A major challenge in main memory based database systems is the issue of backup and recovery. There is the dilemma of what happens to the data in memory when a region is shutdown or if it suffers a crash. Main memory database systems depend on some fast logging mechanisms to permanent storage in order to overcome such issues [13]. This, however, could introduce non-negligible delays into processing that can threaten the performance benefits of a memory based database. For our memory resident co-located index, we rely on the fact that a completely lost index can be recovered by rebuilding it from scratch by scanning the base table region. This, being a local operation, can be accomplished with reasonable performance overhead. Given that crashes are not frequent and only always affect one node, we believe that these recovery costs are reasonable and worth the improved performance during runtime.

However, as the index has to be co-located with each base table region, we need to handle region splits to ensure that the memory resident indices are also split accordingly. This is accomplished by having the coprocessor listen to region splits. Upon the initiation of a region split it will create two new indices from the original in-memory index data structure and persist them into HDFS as separate files. The region server will then shutdown the now split region (which will not exist anymore). The new two daughter regions are then brought up (by possibly different region servers) and the coprocessor instances attached to them will load the corresponding index files from HDFS into main memory.

In similar spirit, we persist the indices to HDFS storage during regular region shutdown so that they can be restored without having to scan the base table region. We do so, because HBase's load-balancing might determine that regions have to be moved. For such moves, it will be faster to persist and transfer the index than re-creating it from scratch at the new region server.

5. EXPERIMENTS

For experimental setup, we used a cluster of nodes each with Intel® Pentium® Dual-Core CPU G2020 @ 2.9GHz , 8GB 1333MHz RAM, 500GB SATA-2 HDD, running Ubuntu 12.04.4 LTS 64-bits. The nodes are connected together using a 1 Gbps switch. Four of the nodes are configured as HBase Region Servers, also running HDFS Data Nodes. One node is setup for running HMaster and HDFS Name Node. Another node was used for running the client processes. We used HBase-0.96 and Hadoop-1.2 for the test setup.

In order to understand the impact of the distribution of secondary indexed attribute's values on performance, we

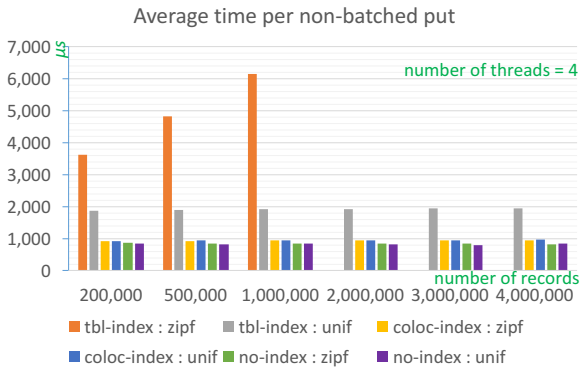

Figure 8: Average time per non-batch put

used both Uniform distribution as well as Zipfian[2] distribution to generate different test datasets. The uniform distribution (over the size of the number of records in the table) ensures that the values are highly distinct. Zipfian on the other hand produces values of a variety of selectivity as the frequency of occurrence of the values vary and hence represents skewed data sets. We used the scrambled Zipfian distribution module in YCSB[3] for our experiments.

To reduce any interference from HBase compactions and java old generation garbage collection [21] on our test metrics, we disabled HBase automatic compactions and increased the jvm heap. Additionally, test metrics were computed by averaging over five runs.

5.1 Non-Batch Writes

To measure the impact on write to base table, we used a client that performed non-batched Put using four threads into an empty table, incrementally adding up to four million row keys and associated secondary index values. We logged the average time taken for every hundred thousand records as the table grew in size to understand the impact of a growing table size. As can be seen in fig. 8, the memory resident co-located index does better than the table-based implementation for both uniform and Zipfian distributions. The performance overhead of co-located index is between 9% to 15% as the table size increases. However, for table-based implementation the performance penalty is almost 125% for uniform distribution. This is to be expected as every Put is now in effect replaced by two Puts. For Zipfian distribution, the performance penalty starts at 300% for 200,000 records and becomes 600% by a million records indicating severe performance issues. This is attributed due to the skewed representation of data in Zipfian distribution resulting in more contention as well as larger secondary index entries which takes even more time to process, compounding the issue. However, this is not an issue for the co-located index as each region processes and updates its secondary index entries locally, causing less contention as the index entries for the same secondary attribute value are distributed across multiple regions. Further, in-memory processing is also faster over HTable, reducing any chance of contention and giving advantage to co-located index.

To measure the scalability of processing, we vary the number of client threads. As can be seen in fig. 9, all except the

[2]https://en.wikipedia.org/wiki/Zipf's_law
[3]https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/

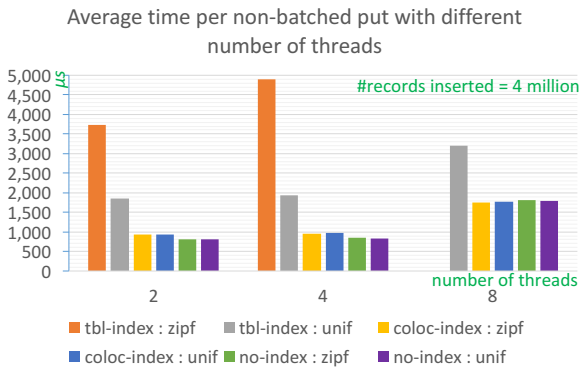

Figure 9: Average time per non-batch put for different number of threads

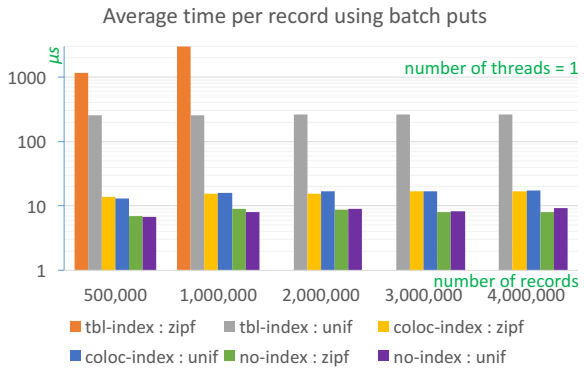

Figure 10: Average time per record for batch put

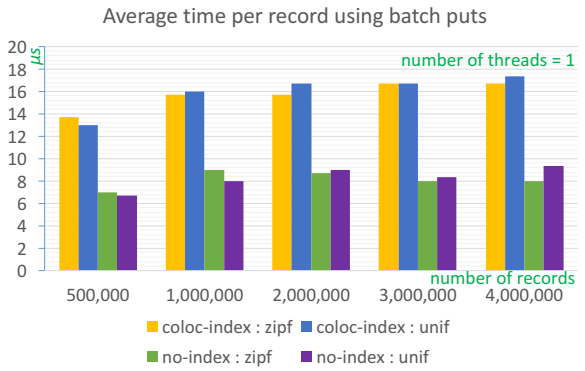

Figure 11: Average time per record for batch put, co-located index vs no index

table-based Zipfian scales well, giving the optimal performance at four threads, where each region server is occupied simultaneously. We observe that the table-based Zipfian fail to scale owing to the contention as discussed before.

5.2 Batch Writes

Further, we performed the same experiment, this time with batched Put, which is the most efficient way of data loading in HBase. We used a single thread and batches of 100,000 records. From the results in fig. 10, we can see that the co-located index does far better than table-based indexing. Co-located indexing still has about 95% penalty. This can be expected because, with batched Puts, inefficient network traffic is reduced, and the only performance bounds is

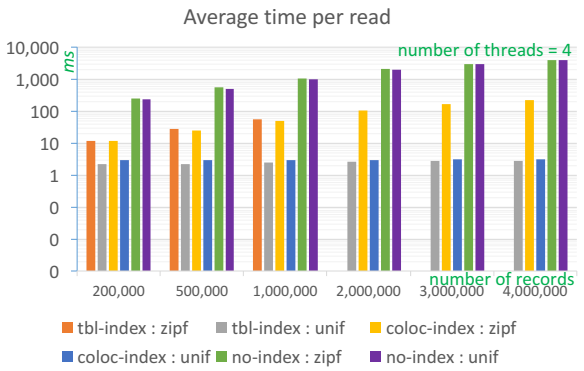

Figure 12: Average time per read using secondary index

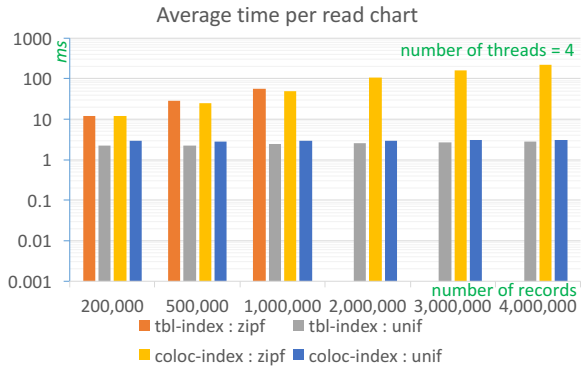

Figure 13: Average time per read using secondary index

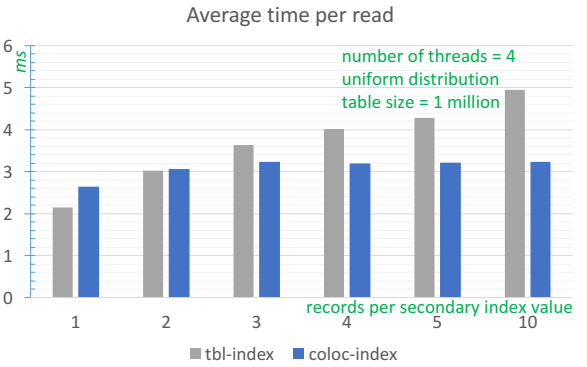

Figure 14: Average time per read using secondary indexed attributes at varying selectivity

by the update to the base table and index itself. While table-based Zipfian continues to show deteriorated performance, what is intriguing is that the performance penalty of table-based index with uniform distribution is also very severe. The reason for this is that, though the client uses batch mode for performing Puts, the coprocessor architecture of HBase results in the coprocessors being invoked once per each Put, dampening the benefits of batched approach. The coprocessor is therefore forced to update the index HTables using individual Puts. Fig. 11 shows the same results without including table-based implementation for better scalability.

5.3 Point Reads Using Secondary Indices

Point reads are performed using the indices created on secondary attributes and compared against the naïve HBase

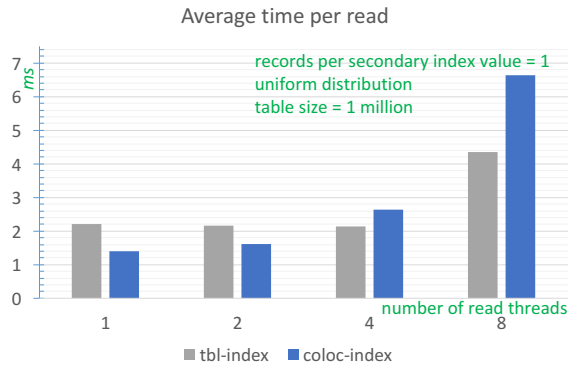

Figure 15: Average time per read using secondary indexed attributes with varying number of threads

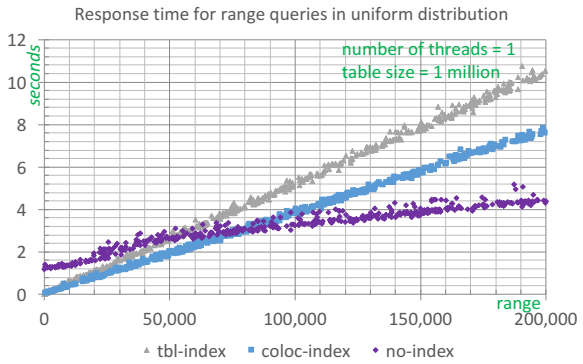

Figure 16: Response time for range queries in uniform distribution

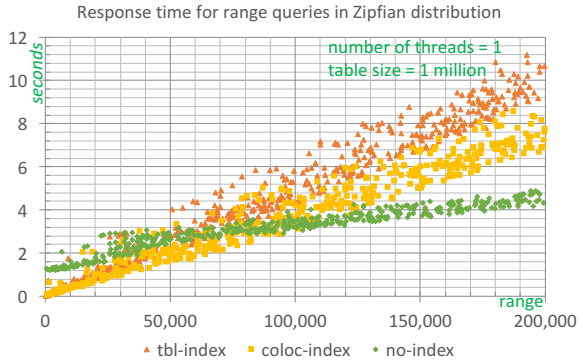

Figure 17: Response time for range queries in Zipfian distribution

alternative, the scan operation. The scan operation results in all of the region to be read. We used tables with different number of records to better measure the variance in performance as a function of table size. A client with 4 threads is used to issue the read requests. The values for secondary indexed attributes are drawn from the same distribution used to load them into the tables to perform random reads.

The results are shown in fig. 12. For better scale, just the comparison of the two indexing approaches is shown in fig. 13. We can see that all the index implementations provide better performance. For uniform distribution, the memory resident, co-located index performance increases from 77 times to 1280 times as the table size grows from 200,000 to 4 million. The performance benefits for co-located index for Zipfian stays around 21%. This is because with Zipfian, there are more records for popular values, and this results in more processing and data transfer. However, even in such cases the benefit of having the index is very evident.

Table-based Zipfian shows the least improvement, owing to the fact that a large number of records will have to be read from all the region servers. Hence retrieving the row keys first and then querying the base records using them turns out to be costly across the network hops. However, table-based index seems to do better with Uniform distribution. It gives 100 to 1400 times better performance, outdoing even the memory resident co-located index implementation.

To analyze this further, we next ran the test case for uniform distribution, varying the number of records per secondary index value. The results shown in fig. 14 shows that as the number of records increase, the table-based index looses its advantage over the co-located one. This is because with increase in number of records to be retrieved from the base table, the table-based implementation will likely has to interact with all the region servers, a scenario in which the co-located index will do better due to the reduced number of hops in the overall read request due to its broadcast nature.

We also analyzed the impact of concurrency by varying the number of read threads while keeping the number of base table records per secondary index value at 1. From the results in fig. 15, we can see that co-located index can do better at lower concurrency, but as the number of threads increases, table-based index starts doing better. This is because, at small workloads, the overhead of broadcast in co-located implementation does not weigh-in into performance, while it also benefits from the reduced request rounds. But at higher concurrency, this overhead starts to come into play.

Therefore on doing detailed analysis on the performance of read queries, one can conclude that table-based implementations are better choice for indexed attributes with near unique distribution, while co-located implementations provides better performance when the secondary index distribution tends to be more non-unique.

5.4 Range Queries using Secondary Indices

To test range queries, we chose a single region setup in the interest of fairness, as HBase scans one region at a time in contrast to our parallel approach. Fig. 16 shows the performance for uniform distribution and fig. 17 shows the same for Zipfian distribution. In both distributions, the memory resident co-located index seems to outperform the table-based implementation. This is in concurrence to our observation in section 5.3 where the co-located index fairs better when there are many records to be returned.

Further, as the range increases, the performance advantages of secondary indices decrease and beyond some range, it becomes an overhead to use the index. This threshold seems to appear at about the range of 80,000 (8% of records in the table.). Beyond this range, scanning the base table region directly is beneficial over using the secondary index.

6. RELATED WORK

As stated in section 3.3, Teradata follows a co-located indexing approach for non-unique attributes and a re-distribution of index entries for unique attributes. However, in Teradata, row distribution (for non co-located index entries as well as

base table records) is accomplished via automatic hashing of the attribute values [25]. While this leads to ease of implementation and maintenance of table and index structures, the downside is there is no locality of associated attribute values due to hashing, and operations such as range searches will have to be performed in all the nodes.

ITHBase [3] is an open source implementation developed by modifying base HBase source code to include transaction and secondary index support. Their implementation is similar to our table-based indexing solution.

IHBase [2] is an in-memory secondary indexing solution for HBase, quite similar to our own in-memory co-located indexing solution. By modifying core HBase, only the portion of the data in disk is indexed and the contents of the memstore is searched completely. The index is built from scratch by scanning the region every time it is brought online.

Culvert [1] is a secondary index implementation similar to our table-based approach. However, it seems like they do their index table updates via modified client side code which needs to be used to perform data loading.

Lily [4] is a cloud-scale repository for social content applications that uses HBase for data storage and Solr [6] for indexing needs. Their indexing engine therefore resides outside the HBase ecosystem.

For the sake of brevity, we have covered only the variants of HBase indexing implementations that are popular in literature. In our implementation, we focused on leveraging the flexibility of the coprocessor framework over the approaches that required modification to core HBase classes. This has the advantage that existing applications need not be migrated to perform index maintenance. Further, our analysis demonstrates that either of the indexing approaches could be the optimal solution depending on the data distribution.

7. CONCLUSION & FUTURE WORK

In this paper we described the challenges associated with indexing in distributed DBMS and provided two implementations for HBase that works with less intrusion into the core HBase source code. Our indexing solutions can work with point queries as well as range queries. Further, we provided a very detailed analysis of how different data distributions warrant different indexing approaches and demonstrated a case for both implementations. Our results show that there is clearly a benefit to having secondary indices in HBase, and that they can be often built with reasonable performance overhead. Although there has been some prior works to achieve secondary indexing in HBase, our work have been more detailed and insightful about the various alternatives and clearly shows that there is no one-stop solution to secondary indexing needs in HBase.

For our future work, we are exploring on how to use EndPoint coprocessors to re-build secondary indices for batch processing-only type of workloads where real-time consistency of index structures are not necessary. This approach is also common in large relational databases where the indices are dropped during bulk loads and rebuilt after the data loads are completed. A key challenge would be how to tackle the client reads while indices are being rebuilt in a less intrusive manner.

8. REFERENCES

[1] Culvert. https://github.com/booz-allen-hamilton/culvert.

[2] IHBase. https://github.com/ykulbak/ihbase.

[3] ITHBase. https://github.com/hbase-trx/hbase-transactional-tableindexed.

[4] Lily. http://www.lilyproject.org/.

[5] Secondary Indexes and Alternate Query Paths. http://hbase.apache.org/0.94/book/secondary.indexes.html.

[6] Solr. http://lucene.apache.org/solr/.

[7] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-Oriented Database Systems. *VLDB*, pages 1664–1665, 2009.

[8] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM SIGMOD*, pages 967–980, 2008.

[9] D. Borthakur. HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2008.

[10] G. Bracha. Generics in the Java programming language. *Sun Microsystems*, pages 1–23, 2004.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, pages 4:1–4:26, June 2008.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[13] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowledge and Data Eng.*, pages 509–516, Dec 1992.

[14] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[15] T. Johnson and D. Sasha. The Performance of Current B-tree Algorithms. *ACM Tran. on Database Systems*, pages 51–101, 1993.

[16] A. Khurana. Introduction to HBase Schema Design. *Networked Syst*, 37:29–36, 2012.

[17] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[18] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, 2010.

[19] P. L. Lehman et al. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. on Database Systems*, pages 650–670, 1981.

[20] A. P. Mingjie Lai, Eugene Koontz. Coprocessor Introduction. https://blogs.apache.org/hbase/entry/coprocessor_introduction, 2012.

[21] Oracle Corporation. Java Garbage Collection Basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[23] V. Srinivasan and M. J. Carey. Performance of B+ Tree Concurrency Control Algorithms. *VLDB*, pages 361–406, 1993.

[24] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[25] Teradata Corporation. Introduction to Teradata. www.teradata.com/workarea/downloadasset.aspx?id=17947, 2010.