# A Comparative Analysis of Materialized Views Selection and Concurrency Control Mechanisms in NoSQL Databases

Ashish Tapdiya Vanderbilt University Email: ashish.tapdiya@vanderbilt.edu Yuan Xue Vanderbilt University Email: yuan.xue@vanderbilt.edu Daniel Fabbri Vanderbilt University Email: daniel.fabbri@vanderbilt.edu

Abstract—Increasing resource demands require relational databases to scale. While relational databases are well suited for vertical scaling, specialized hardware can be expensive. Conversely, emerging NewSQL and NoSQL data stores are designed to scale horizontally. NewSQL databases provide ACID transaction support; however, joins are limited to the partition keys, resulting in restricted query expressiveness. On the other hand, NoSQL databases are designed to scale out linearly on commodity hardware; however, they are limited by slow join performance. Hence, we consider if the NoSQL join performance can be improved while ensuring ACID semantics and without drastically sacrificing write performance, disk utilization and query expressiveness.

This paper presents the Synergy system that leverages schema and workload driven mechanism to identify materialized views and a specialized concurrency control system on top of a NoSQL database to enable scalable data management with familiar relational conventions. Synergy trades slight write performance degradation and increased disk utilization for faster join performance (compared to standard NoSQL databases) and improved query expressiveness (compared to NewSQL databases). Experimental results using the TPC-W benchmark show that, for a database populated with 1M customers, the Synergy system exhibits a maximum performance improvement of 80.5% as compared to other evaluated systems.

Index Terms—Transaction processing, materialized views, NoSQL databases, performance evaluation

## I. INTRODUCTION

Application development with relational databases as the storage backend is prevalent, because relational databases provide: a formal framework for schema design, a structured query language (i.e., SQL) and ACID transactions. As applications gain popularity and the database system reaches its resource limits, the architecture must scale up to ensure end-to-end response time (RT). Relational databases are well suited for vertical scaling; however, vertical scaling has known limitations and requires expensive hardware. On the contrary, the new brand of NewSQL and NoSQL databases are known for their ability to scale out linearly [1], [2], [3]. Hence, as the data size and the resource demands increase, application designers can consider transitioning from their relational database to a NewSQL/NoSQL database. Recently Facebook [4] and Netflix [5] transformed part of their relational databases to HBase [6].

A short version of this paper appeared in IEEE Cluster Conference 2017. Available Online: http://ieeexplore.ieee.org/document/8048951/

NewSQL architectures enable a database to scale out linearly while providing ACID transaction guarantees. However, their schema design requires careful consideration when choosing partition keys, since joins are restricted to partition keys only [7], resulting in limited query expressiveness. Similarly, NoSQL databases can also scale out linearly, but are limited by slow join performance due to the distribution of data across the cluster and data transfer latency, which has also been identified in previous work [8]. Thus, while NewSQL and NoSQL systems allow data stores to scale, their designs sacrifice query expressiveness and join performance, respectively. More generally, there exists a design space that makes trade-offs between performance, ACID guarantees, query expressiveness and disk utilization.

This paper considers if NoSQL join performance **can** be improved while ensuring ACID semantics and **without** drastically sacrificing write performance, disk utilization and query expressiveness. One option for improving the performance of NoSQL workloads is materialized views (MVs), which precompute expensive joins [9], [10]. However, deploying MVs on top of a NoSQL store does not guarantee consistency as atomic key based operations allow for the MV's data to be stale relative to the base table [6], [11], [12]. Hence, additional concurrency controls such as locking or multi-versioning are required to ensure data consistency.

Standard concurrency control methods, such as locking or multi-versioning, can provide ACID semantics for NoSQL stores with materialized views, but induce performance degradation (i.e., by grabbing many locks or checking multiple versions, respectively) because the concurrency control mechanism and MVs selection mechanism are not designed in tandem. Instead, this paper considers a synergistic design space in which the concurrency control mechanism and MV's selection mechanism operate together such that only a single lock is grabbed per transaction. The proposed system relies on the hierarchical structure of relational data and the workload to inform the views selection mechanism, which can then be leveraged to grab a single lock for MVs and base tables.

In this work, we present the Synergy system that leverages MVs and a light-weight concurrency control on top of a NoSQL database to provide for scalable data management with familiar relational conventions and more robust query

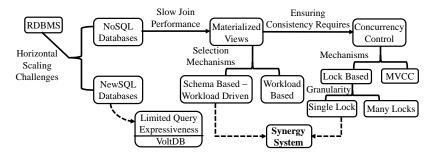


Fig. 1. Design choices and decisions in the Synergy System.

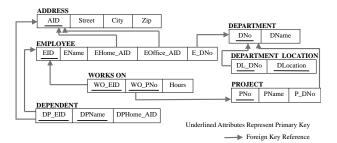


Fig. 2. Relations in the Company schema.

expressiveness. Figure 1 presents the design decisions for MVs selection and concurrency control mechanisms in the Synergy system. Synergy harnesses databases' hierarchical schemas to generate candidate MVs, and then uses a workload driven selection mechanism to select views for materialization. To provide ACID semantics in the presence of views, the system implements concurrency controls on top of the NoSQL database using a hierarchical locking mechanism that only requires a single lock to be held per transaction. The Synergy system provides ACID semantics with the read-committed transaction isolation level. Our contributions in this work are as follows:

- We present the design of Synergy system that trades slight write performance degradation and increased disk utilization for faster join performance (compared to standard NoSQL databases) and improved query expressiveness (compared to NewSQL databases).
- We propose a novel schema based–workload driven materialized views selection mechanism.
- We implement and evaluate the proposed system on an Amazon EC2 cluster using the TPC-W benchmark.
- We compare and contrast the performance of Synergy system with four complementary systems.

## II. BACKGROUND

We first review the concepts of a relation, index and schema which are common to both SQL and NoSQL data models. Then, we present a model for the database workload. Finally, we provide an overview of the data store used and its associated SQL skin.

## A. Relation, Index and Schema Models

**Relation**– A relation R is modeled as a set of attributes. The primary key of R denoted as PK(R), is a tuple of attributes that uniquely identify each record in R. The foreign key of R denoted as FK(R), is a set of attributes that reference another relation T. A relation can have multiple foreign keys, hence let F(R) denotes the set of foreign keys of R.

**Index–** In this work we utilize covered indexes that store the required data in the index itself. An index X on a relation R denoted as X(R), is modeled as a set of attributes (s.t. X(R)  $\subset$ R). Let  $X^{tuple}(R)$  denotes a tuple of attributes that the index is indexed upon (s.t.  $X^{tuple}(R) \subset X(R)$ ). The key of an index is a union of attributes in tuples  $X^{tuple}(R)$  and PK(R), in that order. Since a relation can have multiple indexes, let I(R) denotes the set of indexes on R.

**Schema**– Using the previous definitions of a relation and an index, a database schema S is modeled as a set of relations and the corresponding index sets,  $S = \{R_1, I(R_1), R_2, I(R_2),..., R_n, I(R_n)\}$ , where *n* represents the number of relations in the schema. We use an example Company database for the purpose of exposition. Figure 2 depicts the relations in the Company database schema.

## B. Modeling Workload

A database workload  $W = \{w_1, ..., w_m\}$  is modeled as a set of SQL statements, where *m* is the number of statements.

## C. HBase Overview

We use HBase [6] for the purpose of exposition and experimentation in this work. It is a column family-oriented distributed database modeled after Google's Bigtable [11]. HBase organizes data into tables. A table consists of rows that are sorted alphabetically by the row key. HBase groups columns in a table into column families such that each column family data are stored in its own file. A column is identified by a column qualifier. Also, a column can have multiple versions of a data sorted by the timestamp.

The HBase data manipulation API comprises of five primitive operations: Get, Put, Scan, Delete and Increment. The Get, Put, Delete and Increment operations operate on a single row specified by the row key. HBase provides ACID transaction semantics with read-committed isolation level.

# D. Phoenix Overview

Apache Phoenix [13] is a SQL skin on top of HBase. The client embedded JDBC driver in Phoenix transforms the SQL query into a series of HBase scans and coordinates the execution of scans to generate a standard JDBC result set. The default transaction semantics in Phoenix with base tables only is same as HBase; however, recent integration with Tephra [14] enables multi-statement transactions in Phoenix through MVCC. Note, the MVCC transaction support in Phoenix can be turned on/off by starting/stopping Phoenix-Tephra transaction server. Next, we describe the mechanism to perform a **baseline transformation** from a relational to a NoSQL database.

**Baseline Schema Transformation** – A relation R becomes a relation R' in NoSQL schema with the same set of attributes as R. The row key of R' is a delimited concatenation of the value of attributes in PK(R). Similarly, an index X(R) on a relation R becomes a relation X(R') in NoSQL schema with the same set of attributes as X(R). The row key of X(R') is a delimited concatenation of the value of attributes in the key of X(R). Note that in NoSQL, both for a relation and an index, we assign all attributes to a single column family.

**Baseline Workload Transformation** – Each read statement from the relational workload is added to the NoSQL workload. Each write statement for a relation R that specifies each key attribute in the WHERE clause is added to the NoSQL workload.

#### **III.** CHALLENGES AND DESIGN CHOICES

Joins are expensive in a NoSQL database due to the distribution of data items across different cluster nodes. It is well understood that MVs improve join performance by precomputing and storing results in the database [9], [10], [15]. This observation is verified with TPC-W micro-benchmark which shows that scanning a MV is significantly faster than the join performance (see Section IX-B for experiment details). Thus, we consider how to incorporate MVs into a NoSQL store, while ensuring consistency.

1) Implication of Materialized Views: NoSQL databases are generally limited to key-based single row operations [6], [11], [12]. Hence, to ensure the ACID semantics in the presence of MVs, view maintenance and concurrency controls are required to ensure consistency between the MVs and base tables. The design choices for concurrency control mechanisms include multi-versioning, locking and timestamp ordering. While multi-versioning may seem like a nature fit given HBase and other NoSQL system's temporal key component (i.e., cell values are composed of a row-key, column family, column and time stamp) [11], [12], experimental results show that getting and checking additional rows' timestamps decreases performance. Therefore, this result motivates a lock-based concurrency control mechanism to attain the read committed isolation level.

2) Lock Number and Granularity: Row level locks and database locks represent the two ends of the locking mechanism spectrum. Database locks degrade system throughput

since only a single transaction can access the database at a time. Similarly, acquiring row level locks on individual base tables can be expensive in the presence of MVs in a NoSQL database, since the system may need to acquire a large number of locks for complex queries. Experimental results show that for a modest number of 100 locks, the time to acquire and release locks is 1.3x the response time of the most expensive write transaction in the proposed system (see Section IX-C and Section IX-D4). This observation motivates minimizing the number of locks required per transaction.

3) View Selection Challenges: The types of MVs that are allowed impact the data store performance in varying ways. Purely workload based MVs selection mechanisms [16] (schema relationships are oblivious) can result in optimal read performance by allowing for the materialization of a maximum number of joins in the workload (i.e., views constructed with many-to-many joins or non-foreign key joins). While this approach is well suited for OLAP workloads, it can degrade write performance and increase disk utilization and transaction management costs for the OLTP workloads, especially in a distributed database. In contrast a schema aware-workload driven MVs selection mechanism limits the type of views allowed, resulting in sub-optimal read performance. However, this approach prevents high storage costs and shifts of the bottleneck from read to the write performance. Given the design goal to hold a single lock per transaction across base tables and MVs, this observation motivates us to not allow views with many-to-many joins or joins that do not have key relationships.

## A. Design Decisions

For the Synergy system, we make the following design decisions based off of our analysis of the TPC-W benchmark, which contains many key/foreign-key equi-joins. First, we develop a concurrency control mechanism that leverages the schema's relational hierarchy, grabs one lock per transaction and provides the read committed isolation level. Second, in cooperation with our concurrency control mechanism, the system only materializes key/foreign-key equi-joins, does not materialize joins across many-many relationships, and each base relation may only be assigned to a single relational hierarchy for materialization (so that a single lock must be acquired per transaction). We believe the synergistic design decisions between the concurrency control and view selection mechanism provides for a novel architecture and substantially differentiates this work from previous works on materialized view selection.

### **IV. SYSTEM OVERVIEW**

In this section we provide an overview of the **Synergy system**, as depicted in Figure 3. The objective of our system is to design a scalable and high performance NoSQL database while ensuring the ACID semantics.

We first perform a baseline transformation of the input relational database to a NoSQL database using the mechanism described in Section II-D. Due to the slow join performance

TABLE I. Qualitative comparison of NoSQL, NewSQL and Synergy systems.

	Scalability Query Expressiveness		Transaction Support	Disk Utilization
NoSQL (HBase)	Linear scale out	SQL	ACID with Snapshot Transaction Isolation	Higher than NewSQL
NewSQL (VoltDB)	Linear scale out	SQL with joins limited to partition keys	ACID with Serializable Transaction Isolation	Lowest
Synergy	Linear scale out	SQL with MVs limited to Key/Foreign-Key joins	ACID with Read Committed Transaction Isolation	Highest

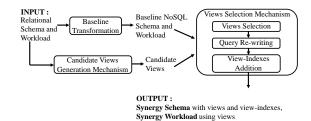


Fig. 3. Synergy system overview.

in the baseline transformed database system, we decide to use MVs. We use the **candidate views generation mechanism** to create a list of potential views to materialize based on the database's hierarchical structure. Next, we use a **workload driven view selection mechanism** to select views from the candidate set. Then, we re-write the workload using selected views as needed. To ensure high read performance, we supplement the schema with additional view-indexes. To ensure ACID semantics in the presence of views, we implement a concurrency control layer on top of HBase (as described in Section VIII), which is able to grab a single lock per transaction, while providing the read committed transaction isolation level.

**System Limitations** – The Synergy system only materializes key/foreign-key equi-joins. In addition, Synergy system is restricted to single SQL statement transactions. In agreement with our design decision of single lock per transaction, write statements that do not specify all key attributes and affect multiple base table rows are not supported. The Synergy system does not enforce foreign key constraints. The transaction isolation level in the Synergy system is limited to read committed. In addition, the Synergy system can only be used with NoSQL data stores that trade availability for strong consistency in presence of network partition (CP model from the CAP theorem [17]).

### V. GENERATING CANDIDATE VIEWS

In this section we present a mechanism to create candidate views for the materialization of equi joins in the workload. We observe that the joins are slow in a NoSQL database (see Section III). Hence, materializing the joins in the workload as views can improve the query performance. We harness the schema's structure to identify the candidate views, in particular the key/foreign-key relationships. We begin by presenting formal definitions for schema relationships and views.

We assume that the input schema S is normalized and free from both simple and transitive circular references, to limit the scope of this work. We model the relationships in S as a directed graph G=(H,E). The vertices in G represent the relations in S and edges encode the key/foreign-key relationship between relations. An edge exists between relations  $R_i$  and  $R_j$ , if they are related as described by:

Definition 1 (Schema Relationships): The relationship between relations  $R_i$  and  $R_j$ , denoted as  $R_i \leftarrow R_j$ , exists iff  $FK_k(R_i)$  references  $PK(R_j)$ , where  $FK_k(R_i) \in F(R_i)$ 

Figure 4(a) depicts the schema graph corresponding to the relations in the Company database schema in Figure 2. Next, we define an edge and a path in the schema graph.

Definition 2 (Edge in Schema Graph): A directed edge  $e_i$ in a schema graph from a relation  $R_i$  to a relation  $R_j$  is represented as a (*PK*,*FK*) tuple where PK is the primary key of  $R_i$  and FK is the foreign key of  $R_j$ .

Definition 3 (Path in Schema Graph): A path between relations  $R_i$  and  $R_j$  in a schema graph is modeled as an alternating sequence of relations and directed edges between the relations,  $[R_i,e_i,...,e_{j-1},R_j]$ . The alternating sequence begins and ends in a relation.

Database schemas have a hierarchical structure; hence, we can choose a set of relations in the schema graph as roots to create rooted trees. Next, we define a rooted tree.

Definition 4 (Rooted Tree): A rooted tree T is a directed graph composed of a subset of nodes and edges from the schema graph in which there exists a root node, and unique paths from the root node to each non-root node.

We use rooted trees to identify the candidate views. Next, we define a candidate view.

Definition 5 (Candidate View): A candidate view V is a path in a rooted tree. A view is stored physically as a relation. The attributes of V is a set union of attributes of relations in V and the key of V denoted as PK(V) is the key of the last relation in the view. Also, a view-index has the same definition and semantic as a table index.

## A. Roots Selection

Each view has a single root. The set of roots Q for a schema S is a subset of relations in S. Q can either be provided by the database designer or it can be learned in an automated manner. In this work, we assume that the database designer provides Q. Note that the automated selection of roots is a separate problem and can be addressed independently.

#### B. Candidate Views Generation Mechanism

The goals of the candidate views generation mechanism are as follows:

• Assign each non-root relation in the schema graph to at most one root. This enables us to hold a single

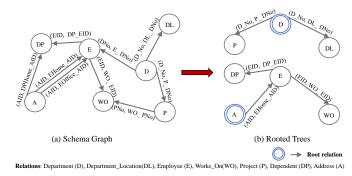


Fig. 4. Input and output of the candidate views generation mechanism for the Company database with roots set  $Q_{company} = \{Address, Department\}$ .

lock on the root relation's row key while ensuring ACID semantics.

• Select a single path between the root and each nonroot relation assigned to it. If there are multiple paths between any pair of relations, then the relationship can be one-many or many-many. However, recall that we define many-many relationship as a join materialization boundary. Therefore, to ensure a one-many relationship, we should select a single path.

1) Mechanism Overview: In this section, we present the overview of the mechanism to generate the candidate views. We first transform the input schema graph into a directed acyclic graph (DAG) to ensure at most one direct path between any pair of relations in the graph. Thereafter, we identify a topological ordering of the relations in the schema DAG. Next, we use the topological order to iteratively examine and assign each non-root relation to a root by selecting a path from the root to the non-root relation. Following the assignment of schema relations to roots, a rooted graph is created for each root relation. Finally, we transform each rooted graph into a rooted tree to ensure a single path between the root and each non-root relation. The output of the mechanism is a set of rooted trees and each unique path in a rooted tree represents a candidate view.

2) Mechanism Description: In this section, we describe the candidate views generation mechanism in detail using our continuing example of the Company database. We use  $Q_{company} = \{Address, Department\}$  as the roots set. In addition, we use a synthetic workload for the purpose of exposition with  $W_{Company} = \{w_1, w_2, w_3\},\$ 

W<sub>1</sub>: Get address details of an employee SELECT \* FROM Employee as e, Address as a WHERE a.AID = e.EHome\_AID and e.EID = ?

 $\mathbf{W}_2$ : Get all the employees and their hours who work in a department.

SELECT \*

FROM Department as d, Employee as e, Works\_On as wo WHERE d.DNo = e.E\_DNo and e.EID = wo.WO\_EID and d.DNo = ?

 $W_3$ : Get all the employees who work a certain number of hours.

SELECT \* FROM Employee as e, Works\_On as wo WHERE e.EID = wo.WO\_EID and wo.Hours = ?

**Heuristic**– During the different steps of the mechanism, we use a heuristic based approach to select a candidate from a set. We choose the **number of overlapping joins** as a simple workload aware heuristic to assign a weight to each candidate. Note that other heuristics can be used seamlessly with the mechanism.

**Input:** Schema graph G, workload W and the roots set Q. **Output:** Set of rooted trees. **Steps:** 

- Transform input graph to DAG: In the first step we transform the input schema graph G into a DAG. We achieve this by selecting and keeping at most one edge between any pair of nodes in the schema graph. We use our heuristic to assign a weight to each candidate edge. Then, we select the edge with maximum weight and remove the rest. For example, we remove the (AID, EOffice\_AID) edge from the schema graph in Figure 4(a) to generate the schema DAG depicted in Figure 5(a).
- 2) **Topologically order relations in the DAG**: Next, we identify a linear ordering of the relations in DAG such that for every directed edge from relation  $R_i$  to  $R_j$ ,  $R_i$  comes before  $R_j$  in the ordering. Figure 5(b) represents a topological ordering of the schema DAG presented in Figure 5(a).
- 3) Assign relations to roots: Next, in the topological order, we examine each non-root relation in the schema DAG and decide upon its assignment to a root by executing the following steps:
  - a) **Identify paths:** We identify paths in the DAG from each root relation to the non-root relation.
  - b) **Select a path:** Next, we utilize our heuristic to assign a weight to each path. Then, we iterate over the paths in the sorted order by weight until we find a path that includes a single root relation and none of the relations on the path are assigned to a root other than the root present in the path.
  - c) Add path: Then, we add the selected path to the rooted graph created for the root in the path.

Figure 5(c) depicts the rooted graphs generated for the Company database.

- 4) **Transform rooted graphs to rooted trees**: Next, we transform the rooted graphs created in step 3 into rooted trees. We first identify a topological ordering of the non-root relations in the rooted graph. We repeat the next step while we have relations left in the topological ordering.
  - a) **Select a Path**: Using the rooted graph we identify paths between the root relation and the last relation

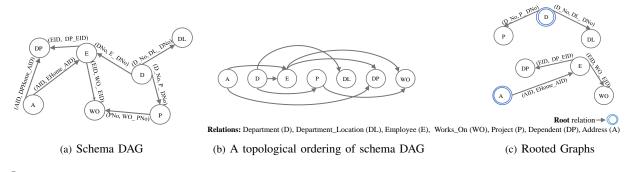


Fig. 5. Intermediate results of the candidate views generation mechanism for the Company database with roots set  $Q_{company} = \{Address, Department\}$ .

in the topological ordering. Next, we assign a weight to each path using our heuristic. Then, we select the path with maximum weight and add it to the rooted tree. Thereafter, we remove all non-root relations in the path from the topological ordering and continue.

Note that in step 3, we examine non-root relations of a schema DAG in the forward topological order to give each non-root relation a chance to be assigned to any root that has a path to it. Conversely, in step 4, we examine non-root relations of a rooted graph in the reverse topological order to keep the paths that will allow materialization of maximum number of joins in the workload. Following the candidate views generation mechanism, a rooted tree is generated for each root in Q. Figure 4(b) depicts the set of rooted trees generated for the Company database.

3) Discussion: The proposed candidate views generation mechanism is a heuristic based approach; hence, does not guarantee materialization of optimal number of joins in the workload. In addition, the usability of generated candidate views for join materialization is dependent on roots selection.

## VI. VIEWS SELECTION MECHANISM

In this section we describe our procedures for views selection from the candidate set and re-writing queries using selected views. Similar to [16], we use a workload driven views selection mechanism. We also illustrate our method for supplementing the schema with additional indexes to ensure query performance.

## A. Views Selection

The high resource requirement and the expensive nature of joins in a NoSQL database (see Section III) provides us with the motivation to materialize as many joins in the workload as possible to ensure low request response times and high system throughput. We use a workload driven approach to select views. We iteratively examine each equi join query in the workload and select views for it. Next, we describe our procedure to select views for a given query.

**Views selection for a Query–**We harness the rooted trees and the query syntax to select views for a query. To illustrate the procedure, we use the example rooted tree and the example query depicted in Figures 6(a) and 6(b) respectively. We begin the procedure with un-marked rooted trees. Then, we use the join conditions in the query to mark the relevant edges and participating relations in the rooted trees. Figure 6(c) depicts the marked edges and relations in the example rooted tree. Next, we examine each rooted tree to identify the views to be selected for the query.

For a given rooted tree, we iteratively choose a path until no new path can be chosen. During each iteration, path selection is done using two rules: 1) all the nodes and edges in the path are marked, and 2) the path starts in a marked node that has no incoming marked edge and ends in either a leaf node or a node that has no outgoing marked edge. Then, we select the chosen path as a view. Next, we un-mark the participating relations of the path and outgoing edges of the participating relations, in the rooted tree. Thereafter, we continue with the next iteration. Figure 6(c) depicts the views selected for the example query.

**Final View Set-** After processing the entire workload, we add the set of all selected views to the schema.

**Limitations**– 1) We select views only for the equi join queries in the workload. 2) Searching the space of all syntactically relevant views is not feasible in practice [16]; hence, similar to [16], our views selection procedure is heuristic based and does not necessarily select the optimal set of views. 3) A views selection procedure that can take advantage of view sharing opportunities across different queries is part of our future work. 4) Currently we do not pass a storage constraint to our views selection algorithm; however, it can be easily adapted to use storage constraint in presence of a cost based query optimizer.

## B. Query Re-writing

Following views selection, we re-write queries using selected views. We iteratively examine each equi join query in the workload and re-write it using the views selected for it. To re-write a query, we replace the constituent relations of a view with the view. In addition, we remove the join conditions for which both participating relations belong to a single view. Figure 6(d) depicts the example query re-written using selected views.

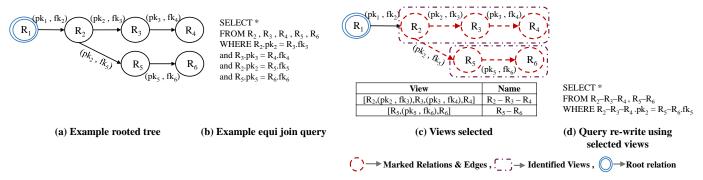


Fig. 6. Illustration of view selection and query re-writing procedure for an example equi join query using an example rooted tree.

## C. Additional View Indexes

Unfortunately, in certain scenarios query execution times can be high despite the use of views. Consider a case in which the query using the view has a filter on an attribute other than the attribute that the view is indexed upon. Then, to prepare the query response, we have to scan the entire view. This can be expensive, depending on the size of the view. Hence, to improve the performance of workload queries that use views, we supplement the schema with additional indexes.

For each view, we examine each conjunctive query that uses this view and decide whether to add a view-index or not. If the query only has filters on one or more view attributes that neither the view nor any of its indexes are indexed upon, then we add a view-index indexed upon a filter attribute to the schema. Note that in this work we do not recommend indexes on base tables and assume that the input schema has necessary base table indexes.

# VII. VIEW MAINTENANCE MECHANISM

In this section, we describe the mechanism for view maintenance as the underlying base tables are updated. For each type of write statement we present: 1) an applicability test to determine if a base table update applies to a view and 2) a tuple construction procedure to prepare tuples for the view update upon a base table update.

## A. Insert Statement

1) Applicability Test: A base table insert for a relation  $R_i$  applies to a view  $V_i$  iff  $R_i$  is the last relation in  $V_i$ 's sequence of relations.

2) Tuple Construction: Insertion into a view upon a base table insert may require reading tuples from the base tables to construct the view tuple. For a base table insert that applies to a view with k relations, we need to read related tuples from k-1 base tables to construct the view tuple. We utilize the key/foreign-key relationships between view relations to sequentially read the base table tuples, starting with relation  $R_{k-1}$  and ending in relation  $R_1$ . Then, we construct the view tuple using previously read tuples and the insert statement. Notice that the time to create a view tuple increases linearly with the number of relations in the view and is independent of the cardinality ratios between the relations.

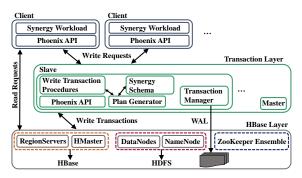


Fig. 7. Synergy System Architecture Overview.

## B. Delete Statement

1) Applicability Test: A base table delete for a relation  $R_i$  applies to a view  $V_i$  iff  $R_i$  is the last relation in  $V_i$ . Note that we do not perform cascading deletes.

2) Key Construction: To delete a view tuple upon a base table *delete*, we use the base table key provided with the delete statement. However, to delete the view index tuple, we need to first construct the index key to issue a delete upon. Hence, we first read the tuple from the view using the base table key in the delete statement. Then, we use the attributes in the read tuple to construct the index key and issue the delete. Notice that the time to construct a view index key is constant.

## C. Update Statement

1) Applicability Test: A base table update for a relation  $R_i$  applies to a view  $V_i$  iff  $R_i$  is in  $V_i$ 's sequence of relations.

2) *Tuple Construction:* Unfortunately, updating the view upon a base table update can be expensive if the view is not indexed on the key of the update statement, since we need to either join the base tables or scan the entire view for the tuple construction. To efficiently prepare view updates, we supplement the schema with additional indexes based on the workload. Due to space concerns, we omit the details.

## VIII. SYSTEM ARCHITECTURE

In this section we describe the Synergy system architecture. The Synergy system comprises of **HBase layer**, **clients** and the **Transaction layer** as depicted in Figure 7. **HBase layer**– The Synergy system harnesses HBase layer as the distributed data storage substrate. The HBase layer comprises of HBase, HDFS and ZooKeeper components. We refer the reader to [6] for the role and description of each component shown in Figure 7.

**Clients–** The clients utilize Phoenix API to execute read and write statements in the workload. A client sends a read request directly to the HBase layer. On the contrary, a write request is sent to the Transaction layer, followed by a synchronous wait for a response.

Transaction Layer- The Synergy system employs the Transaction layer for implementing ACID transaction support on top of the HBase layer. The Transaction layer is a distributed, scalable and fault tolerant layer that comprises of a Master node and one or more Slave nodes. The Slave nodes receive and process write requests from clients. Each slave node has a transaction manager that implements a write ahead log (WAL) for recovery and durability. The WAL is stored in HDFS. Upon receiving a request, the transaction manager first assigns a transaction *id* to the statement and then appends the statement in WAL along with the assigned id. Then, a transaction procedure utilizes Phoenix API to execute the transaction. Finally, a response is sent back to the client. The Master node is responsible for detecting slave node failures and starting a new slave node to take over and replay the WAL of a failed slave node.

## A. Lock Implementation

**Logical Locking**– Recall that we restrict the write workload to statements that specify all key attributes (see Section IV) and decide to employ hierarchical locking as the concurrency control mechanism (see Section III). Hence, to update a row for a relation in a rooted tree, we acquire the lock on the key of the associated row in the root relation. In addition, since each relation is part of at most one rooted tree, we hold a single lock per write operation.

**Physical Locking**– We implement our locking mechanism through lock tables stored in HBase. We create one lock table per root relation. The lock table key has same set of attributes as the root relation's key and it includes a single boolean column that identifies if lock is in use or not. A lock table entry is created when a tuple is inserted into the root table.

**Discussion**– We implement light weight hierarchical locking mechanism in Synergy by holding a single lock per write operation. As a downside of hierarchical locking, all rows associated with the root key along all the paths are locked which can affect throughput with concurrent requests trying to grab the lock on the same root key. Note that lock management is not the primary contribution of our work. Other transaction management systems like Themis [18], Tephra [14], Omid [19] etc. could also be used.

## **B.** Write Transaction Procedures

Synergy utilizes transaction procedures to atomically update the base table, views and corresponding indexes upon a base table update. For **insert** and **delete** statements, the transaction



Fig. 8. Micro benchmark schema graph.

procedure first acquires the lock on the root key. Then, the base table, applicable views and corresponding indexes are updated using the tuple/key construction procedures described in Section VII. Finally, the lock is released. Note that each transaction inserts/deletes a single row in/from the base table, applicable views and corresponding indexes.

A base table update may require multi-row updates on the materialized view. Now, while a view is being updated upon a base table update, conflict with the concurrent writes is prevented by the locking mechanism; however, a concurrent read may read dirty data. Hence, to facilitate the detection of a dirty read, we mark the data in views and view-indexes before update and un-mark after update. If dirty data is read in a transaction, then the read is restarted. The **update** transaction is a 6-step procedure: 1) We first acquire a lock on the root key. 2) Then, we read all the rows that need to be updated. 3) Next, we mark all the rows that need to be updated. 4) Then, we issue a sequence of updates. 5) Next, we un-mark all the updated rows. 6) Finally, we release the lock.

The plan generator component (see Figure 7) in the Synergy transaction layer auto generates the execution plan for each write transaction.

## C. Transaction Isolation Level

The Synergy system is restricted to single statement transactions. In addition, Synergy does not support queries in which a relation is used more than once due to potential dirty reads. The Synergy system provides **ACID semantics** with **read committed** transaction isolation level, which is also the default transaction isolation level for PostgreSQL [20].

A single row is inserted/deleted into/from the base table, applicable views and corresponding indexes upon a base table **insert/delete**. In addition, to answer a query either a table is used directly or a view involving the table is used but not both. Hence, a reader either reads the entire row or the row is absent from the read result set. This enables read committed behavior for insert and delete statements.

Recall that the system marks the rows to be updated in a view as dirty before issuing updates, and if a concurrent *scan* reads a dirty row, the scan is restarted. Hence we modify the scan behavior to check for marked rows in the scanned resultset and **re-scan** if a marked row is present. This ensures that **update** statement preserves the read committed semantics.

Note that the **read committed** semantics are preserved during a **failure scenario**, since the base table lock is held until the system recovers from the failure.

## IX. EXPERIMENTAL EVALUATION

In this section we first describe our experiment environment. Next, we use a TPC-W micro-benchmark to evaluate the join performance in HBase. Thereafter, we profile the performance Q1: Get all the customers and their orders Using base tables, SELECT \* FROM Customer as c, Order as o WHERE c.c\_id = o.o\_c\_id Using view, SELECT \* FROM Customer-Order Q2: Get all the customers, their orders and the constituting order lines. Using base tables, SELECT \*

FROM Customer as c, Order as o, Order line as ol WHERE c.c\_id = o.o\_c\_id and o.o\_id = ol.ol\_o\_id Using view,

SELECT \* FROM Customer-Order-Order line

Fig. 9. Micro-Benchmark Workload.

overhead of two phase row locking in HBase. Finally, we evaluate the performance of Synergy system and compare it with four other systems using the full TPC-W benchmark.

## A. Experiment Environment

1) Testbed: Amazon EC2 represents our experiment environment. We create an eight node cluster using m4.4xlarge virtual machine (VM) instances. Each instance is configured with 16 vCPU's, 64GB RAM and 120 GB SSD elastic block storage (EBS), running Ubuntu 14.04.

*HBase, HDFS and Zookeeper:* The HDFS NameNode, the HBase HMaster, and the ZooKeeper server processes run on one instance. We designate five instances as slaves, each running the HDFS DataNode and the HBase RegionServer processes. We use Hadoop v2.6.5, HBase v1.2.4.

*Synergy and Phoenix:* We dedicate one instance to host a Synergy transaction layer slave and the Phoenix-Tephra server. Synergy transaction layer master is hosted on the same node that hosts HBase and HDFS masters. We use Phoenix v4.8.2.

*VoltDB:* We create a five instance VoltDB (v6.8) cluster by hosting a VoltDB daemon on each instance that is also hosting the HDFS DataNode and the HBase RegionServer processes.

*Client:* We reserve one node as client to drive the workload for each system.

2) Performance Metric: The request response time represents our performance metric, denoted as  $\tau$ . We measure  $\tau$  in the client.

## B. Micro Benchmark Evaluation

We use a TPC-W micro-benchmark to evaluate the join performance in HBase.

1) Schema and Workload: The micro benchmark schema comprises of three relations from the TPC-W benchmark: Customer, Order and Order\_line. Customers can have one or more orders and each order can have one or more order lines. Figure 8 depicts the schema graph for the benchmark schema. Next, to evaluate the join performance, we create a synthetic workload comprising of two foreign key equi-join queries: Q1 (*Customer,Order*) and Q2 (*Customer,Order,Order\_line*).

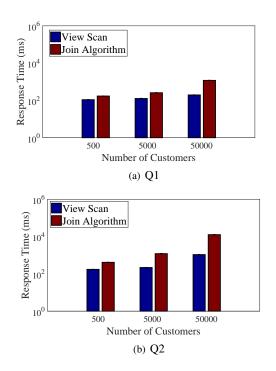


Fig. 10. Micro benchmark results to show that performance of join algorithms is slow in HBase. Y axis is drawn at log scale.

	Number of Locks				
	10	100	1000		
Overhead (in ms)	342	571	2182		

Fig. 11. Experiment to show overhead associated with two phase row locking in HBase.

A join query can be evaluated using two different approaches: 1) using a join algorithm that combines the matching tuples from the specified tables and 2) scanning pre-computed and stored results from a materialized view. Hence, to compare the join algorithm performance with the view scan performance, we materialize the joins in the workload as views. Customer-Order and Customer-Order-Order\_line represent the MVs corresponding to the join queries Q1 and Q2 respectively. Figure 9 presents the workload queries written using base tables and MVs.

2) Experiment Setup and Results: Each experiment is characterized by the database scale and the join query. We set the cardinality ratio between relations as 1:10. We scale the database by increasing the number of customers in multiples of 10, starting at 500. For each database scale, we major compact both base tables and views after database population. Section IX-B1 presents the join queries in the workload. We repeat each experiment 10 times and report the mean and the standard error of response time. Figure 10 depicts the experiment results with Y axis drawn at log scale.

For the database populated with 50K customers, view scan is 6x and 11.7x faster than the join algorithm for queries Q1

and Q2 respectively. In conclusion, micro-benchmark results show that the join algorithm performance is slow in HBase, providing the motivation for join materialization.

## C. Locking Overhead Evaluation

In this experiment, our goal is to evaluate the performance overhead of acquiring and releasing row locks in HBase. We create a single lock table in the HBase layer with two attributes: *id* and *lock\_status*. The *lock\_status* is a boolean column that identifies whether lock is in use or not. We use *checkAndPut* HBase operation in the client node to acquire and release locks. We increase the number of locks in multiples of 10 starting at 10 and measure the overhead in client. We repeat each experiment 10 times and present the mean overhead time. Figure 11 shows the experiment results.

Locking overhead with 100 locks is 1.3x the response time of statement W13 in the Synergy system (see Section IX-D4 and Figure 14); W13 represents the most expensive write transaction in the Synergy system. Also note that 100 represents a modest number of locks for a write transaction, since multiple tables with varying cardinalities may be joined together as a view. *In conclusion, overhead associated with the acquisition and release of row locks represents a major transaction performance bottleneck in HBase, motivating the use of a single lock per transaction.* 

#### D. TPC-W Benchmark Evaluation

1) Benchmark: TPC-W [21] is a transactional web benchmark. It has a two tier architecture including a web tier and a database tier. TPC-W workload includes 14 different types of web requests where each request is modeled as a servlet. Each servlet is in turn composed of one or more SQL statements. We analyze the TPC-W servlets to extract all the SQL statements that can be invoked at the database tier. Extracted set of SQL statements represents our workload.

We exclude a DELETE statement (DELETE FROM *shopping\_cart\_line* WHERE *scl\_sc\_id* = ?) from the workload that may affect multiple base table rows. Phoenix currently does not provide an implementation of the *soundex* algorithm; hence, we exclude two join queries from the workload that use *soundex* algorithm.

The database size  $(DB_{size})$  can be modulated by varying two parameters: the number of customers (NUM\_CUST) and the number of items (NUM\_ITEMS). We set NUM\_ITEMS to 10 \* NUM\_CUST. In addition, we change the cardinality between the Customer and the Orders table from .9 to 10. We populate the database with 1 million customers. For each system that utilizes HBase as the storage layer, we major compact base tables, indexes and MVs after the database population.

2) Systems Evaluated: Synergy: We use  $Q_{TPC-W} = \{Author, Customer, Country\}$  as the roots set to generate views in the Synergy system. We create base tables, selected views and corresponding indexes in HBase. In addition, we create lock tables for each root in  $Q_{TPC-W}$ . We disable the Phoenix-Tephra transaction support.

*MVCC-UA:* To compare our views generation and selection mechanism with [16], we deploy SQL Server 2012 on a single EC2 VM instance. Next, we populate the TPC-W database with 1 million customers and run the TPC-W benchmark queries. Then, we use the SQL Server's database engine tuning advisor to analyze the profiled workload and generate views. We create the generated views along with base tables and indexes in HBase and run the workload with Phoenix-Tephra transaction support (MVCC) enabled.

*MVCC-A:* In addition to the base tables and indexes, we create the views and the view-indexes generated by the Synergy system in HBase and run the workload with Phoenix-Tephra transaction support (MVCC) instead of the specialized transaction support used in Synergy.

*Baseline:* We only create base tables and corresponding indexes in HBase and run the workload with Phoenix-Tephra transaction support (MVCC).

*VoltDB:* A VoltDB table can either be partitioned or replicated. The partitioning column is specified by the user and partitioned tables can only be joined on equality of partitioning column. Now, a table can join with other tables using different columns in different queries of the workload; however, since each table can only be partitioned on a single column, only a subset of workload join queries may work for a partitioning scheme.

To profile the performance of maximum number of joins in the TPC-W benchmark we use **three different partitioning schemes in VoltDB**. However, note that in practice only one partitioning scheme could be used for a database. Also, note that only base tables and corresponding indexes are used in VoltDB.

Figure 13 summarizes the MVs creation and concurrency control mechanisms used in each evaluated system.

3) Performance Evaluation of Joins in the TPC-W Benchmark: Experiment setup- In this set of experiments, we evaluate and compare the join performance across different systems using the join queries in the TPC-W benchmark. Recall that we used three different partitioning schemes in VoltDB to support maximum number of TPC-W joins, using any single partitioning scheme less than 50% of the TPC-W joins are supported.

We evaluate each query10 times and present the mean and the standard error of the recorded response times. See appendix for the specification of join queries in the TPC-W benchmark. Figure 12 presents the experiment results. Note that join queries  $\{Q_3, Q_7, Q_9, Q_{10}\}$  are not supported in VoltDB.

*Discussion*- On an average the join queries in Synergy are 19.5x, 6.2x and 28.2x faster as compared to the MVCC-UA, MVCC-A and Baseline system respectively. The view selection mechanism in the Synergy system selects more MVs as compared to MVCC-UA, resulting in significantly larger join performance benefit. In MVCC-UA, the response time of  $Q_{10}$  is significantly lower than the Baseline system since MVCC-UA utilizes a materialized view for query evaluation. The join performance in Synergy system with specialized con-

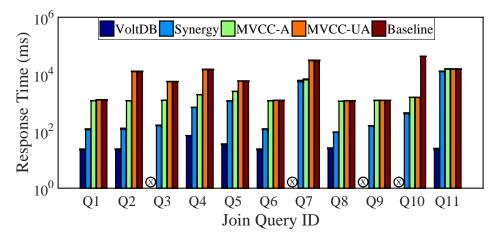


Fig. 12. Evaluation and comparison of join performance across different systems using join queries in the TPC-W benchmark. Y axis is drawn at log scale. Join queries  $\{Q_3, Q_7, Q_9, Q_{10}\}$  are not supported in VoltDB.

Mechanisms	Systems						
Mechanisms	VoltDB	Synergy	MVCC-A	MVCC-UA	Baseline		
Materialized Views Selection	None	Schema Relationships Aware	Schema Relationships Aware	Schema Relationships Un-Aware	None		
Concurrency Control	Single Threaded Partition Processing	Hierarchical Locking	MVCC	MVCC	MVCC		

Fig. 13. Materialized views selection mechanism and concurrency control mechanism used in each evaluated system.

currency control is marginally better than MVCC-A that uses MVCC. The join queries that used views in Synergy are on an average 11x slower than VoltDB (excluding queries that are not supported in VoltDB). In conclusion, join response times in the Synergy system with selected views are significantly lower as compared to MVCC-UA and Baseline system for the benchmark queries. In addition, although the Synergy join performance is slower than VoltDB, Synergy allows for significantly more expressive joins than VoltDB.

4) Performance Evaluation of Write Statements in the TPC-W Benchmark: Experiment setup–In this set of experiments, we aim to evaluate the performance overhead of acquiring/releasing a lock and updating MVs in the Synergy system. In addition, we compare the write statement performance across different systems using the write statements in the TPC-W benchmark. See appendix for the specification of write statements in the TPC-W benchmark. We evaluate each statement 10 times and present the mean and the standard error of the recorded response times. Figure 14 presents our experiment results.

*Discussion*–On an average the write statements in Synergy are 9x, 8.6x and 8.6x less expensive than MVCC-UA, MVCC-A and Baseline system respectively. In Synergy system, the execution time of statements W6 and W11 is significantly lower than the other write statements since the corresponding relation is not part of any views. Although Baseline system does not use any MVs and MVCC-UA utilizes only one materialized view, the statement response times in these systems are high since MVCC adds an overhead of 800-900 ms to each statement's execution time. On an average the write statements in Synergy are 9.4x more expensive than the VoltDB. In conclusion, experimental results show that the use of hierarchical locking in Synergy system significantly reduces the write transaction response times in presence of MVs.

TABLE II. Sum of RT of all the statements in the TPC-W benchmark to quantify trade off between read performance gain and write performance overhead of using MVs in each evaluated system. VoltDB is excluded since it does not support all queries in the benchmark.

	Evaluated Systems				
	Synergy	MVCC-UA	Baseline		
Mean Response Time (in seconds)	33.7	77.4	132.4	173.4	
Standard Error	.03	.02	.06	.07	

5) Performance Comparison of All Evaluated Systems: Experiment setup–In this set of experiments, we evaluate the performance gain and the storage overhead of using MVs in the Synergy system and compare it with the other systems. Note that we exclude VoltDB since it does not support all join queries in the TPC-W benchmark. We evaluate the performance of systems using all the statements in the TPC-W benchmark.

During an experiment, we run each benchmark SQL statement and record its response time. Next, we compute the sum of response time of all statements. We run each experiment 10 times and present mean and standard error of the benchmark response time. Table II presents the experiment results. Table III summarizes the database sizes across different systems.

TABLE III. Database sizes across different evaluated systems.

No. of Customers		Database Size (in GB)					
No. of Customers	VoltDB	Synergy	MVCC-A	MVCC-UA	Baseline		
1M	31.8	92	91.8	45.73	43.8		

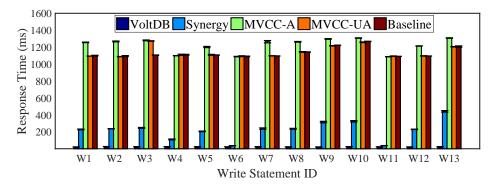


Fig. 14. Performance Evaluation of the write statements in the TPC-W benchmark to exhibit the overhead of lock management and updating views in the Synergy system. Comparison of write statement performance across different systems.

Discussion- Synergy system exhibits a performance improvement of 74.5%, 56.3% and 80.5% as compared to the MVCC-UA, MVCC-A and Baseline system respectively. Conversely, the database size in the Synergy system is 2x, 1x and 2.1x the database size in the MVCC-UA, MVCC-A and Baseline system respectively. Hence, Synergy system trades slight write performance degradation and increased disk utilization for faster join performance. In conclusion, the specialized concurrency control mechanism and the MVs generation mechanism in the Synergy system significantly improve the read performance without shifting the bottleneck to the write performance.

# X. RELATED WORK

Materialized Views. MVs have been studied from multiple standpoints in the SQL domain: view maintenance, view matching, automated views selection, dynamic view maintenance etc. In [22], [23], [24], [8] authors explore the problem of efficient view maintenance in response to the base table updates. The dynamic views [25] introduce storage efficiency by automatically adapting the number of rows in the view in response to the changing workload. The view matching techniques are utilized in query optimization to determine the query containment and the query derivability [9], [10], [15]. In [16], authors propose a workload driven mechanism to automate the task of selecting an optimal number of views and indexes for decision support system applications. The MVs selection and maintenance in a transaction processing NoSQL data store raises novel challenges since most of the existing views selection approaches are oblivious to the relationship between schema relations which can lead to heavy view maintenance costs and can shift the bottleneck from reads to writes. To this end, Synergy proposes a novel, schema relationships aware view selection mechanism.

**Data Partitioning.** Megastore [26], F1 [27] and Elastras [28] harness hierarchical schema structure to cluster related data together and minimize the distributed transactions. On the contrary, Synergy generates MVs utilizing hierarchical schema structure to reduce query run times. In [29], authors automate the task of data partitioning by developing a technique for

automated selection of root relations in a schema. Schism [30] proposes fine grained data partitioning by co-locating related tuples based on workload logs.

Transactions. Transaction support in the majority of the first generation NoSQL stores [6], [12] and Big Data systems [31] is limited to single-keys. G-Store [32] extends HBase to support multi-key transactions in a layer on top using a 2 phase locking protocol. Similar to G-Store, we implement write transactions in a layer on top of HBase. CloudTPS [33] supports multi-key read/write transactions in a highly scalable DHT based transaction layer using optimistic concurrency control (OCC). In [34], authors extend CloudTPS to support consistent foreign-key equi-joins. ecStore [35] provides snapshot isolation using MVCC based on the global timestamp transaction ordering in a decoupled layer on top of an ordered key-value store BATON. ElasTras [28] proposes a novel keyvalue store that implements MVCC based transactions. Percolator [36] extends Bigtable to allow cross-row, cross-table ACID transactions and enables incremental updates to the web index data stored in BigTable. Megastore [26] introduces entity groups as a granule of physical data partitioning and supports ACID transactions with in an entity group. F1 [27] is built on top of Spanner [37] and supports global ACID transactions for the Google AdWords business. In contrast with Spanner, Synergy is limited to single data center use; however, Synergy enables enhanced SQL query expressiveness and does not require sophisticated infrastructure including atomic clocks, GPS etc. The NewSQL databases [3], [7] scale out linearly while ensuring ACID semantics; however, the join support is limited to partitioning keys. The first generation of NewSQL systems required all data to reside in main memory; however, recent work [38] overcomes this limitation by keeping cold data on the disk.

#### **XI.** CONCLUSIONS

In this paper we present the Synergy system, a data store that leverages schema based-workload driven materialized views and a specialized concurrency control system on top of a NoSQL database that allows for scalable data management with familiar relational conventions. Synergy trades slight write performance degradation and increased disk utilization for faster join performance (compared to standard NoSQL databases) and improved query expressiveness (compared to NewSQL databases). Experiment results on a lab cluster using the TPC-W benchmark show the efficacy of our system.

## XII. ACKNOWLEDGEMENT

This work was supported in part by National Science Foundation awards 1526014 and 1150169.

## REFERENCES

- D. Kossmann et al., "An evaluation of alternative architectures for transaction processing in the cloud," in *SIGMOD*, 2010, pp. 579–590.
- [2] B. F. Cooper et al., "Benchmarking cloud serving systems with ycsb," in SoCC, 2010, pp. 143–154.
- [3] R. Kallman et al., "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, pp. 1496– 1499, 2008.
- [4] "Storage infrastructure behind Facebook messages: Using HBase at scale. [online]. available: http://sites.computer.org/debull/a12june/p4.pdf."
- [5] "NoSQL at Netflix. [online]. available: http://techblog.netflix.com/2011/01/nosql-at-netflix.html."
- [6] "HBase. [online]. available: http://hbase.apache.org/."
- [7] "VoltDB. [online]. available: https://voltdb.com/."
- [8] P. Agrawal et al., "Asynchronous view maintenance for vlsd databases," in *SIGMOD*, 2009, pp. 179–192.
- [9] P.-A. Larson and H. Z. Yang, "Computing queries from derived relations," in *VLDB*, 1985, pp. 259–269.
- [10] H. Z. Yang and P.-A. Larson, "Query transformation for psj-queries," in VLDB, 1987, pp. 245–254.
- [11] F. Chang et al., "Bigtable: A distributed storage system for structured data," in OSDI, 2006, pp. 15–15.
- [12] "Accumulo. [online]. available: https://accumulo.apache.org/."
- [13] "Apache Phoenix. [online]. available: http://phoenix.apache.org/."
- [14] "Thepra. [online]. available: http://tephra.incubator.apache.org/."
- [15] J. Goldstein and P.-A. Larson, "Optimizing queries using materialized views: A practical, scalable solution," in SIGMOD, 2001, pp. 331–342.
- [16] S. Agrawal et al., "Automated selection of materialized views and indexes in sql databases," in VLDB, 2000, pp. 496–505.
- [17] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," SIGACT News, 2002.
- [18] "Themis. [online]. available: https://github.com/xiaomi/themis."
- [19] "Omid. [online]. available: https://github.com/yahoo/omid."
- [20] "PostgreSQL: Transaction Isolation. [online]. available: http://www.postgresql.org/docs/9.1/static/transaction-iso.html."
- [21] "TPC-W benchmark. [online]. available: http://www.tpc.org/tpcw/."
- [22] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," in SIGMOD, 1986, pp. 61–71.
- [23] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, "Making views selfmaintainable for data warehousing," in DIS, 1996, pp. 158–169.
- [24] J. A. Blakeley, N. Coburn, and P.-V. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," ACM Trans. Database Syst., vol. 14, no. 3, Sep. 1989.
- [25] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding, "Dynamic materialized views," in *ICDE*, 2007, pp. 526–535.
- [26] J. Baker et al., "Megastore: Providing scalable, highly available storage for interactive services," in CIDR, 2011, pp. 223–234.
- [27] J. Shute et al., "F1: A distributed sql database that scales," Proc. VLDB Endow, pp. 1068–1079, 2013.
- [28] S. Das et al., "Elastras: An elastic, scalable, and self-managing transactional database for the cloud," ACM Trans. Database Syst., vol. 38, no. 1, pp. 5:1–5:45, Apr. 2013.
- [29] B. Liu et al., "Automatic entity-grouping for OLTP workloads," in *ICDE*, 2014.
- [30] C. Curino et al., "Schism: A workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*, pp. 48–57, 2010.
- [31] S. Alsubaiee et al., "Asterixdb: A scalable, open source bdms," Proc. VLDB Endow., pp. 1905–1916, 2014.

- [32] S. Das et al., "G-store: A scalable data store for transactional multi key access in the cloud," in SoCC, 2010, pp. 163–174.
- [33] Z. Wei, G. Pierre, and C.-H. Chi, "CloudTPS: Scalable transactions for Web applications in the cloud," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 525–539, Oct-Dec 2012.
- [34] —, "Scalable join queries in cloud data stores," in *CCGrid*, 2012, pp. 547–555.
- [35] H. T. Vo et al., "Towards elastic transactional cloud storage with range query support," *Proc. VLDB Endow.*, pp. 506–514, 2010.
- [36] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in OSDI, 2010, pp. 1–15.
- [37] J. C. Corbett et al., "Spanner: Google's globally-distributed database,"
- in OSDI, 2012, pp. 251–264.
  [38] J. DeBrabant et al., "Anti-caching: A new approach to database management system architecture," *Proc. VLDB Endow.*, pp. 1942–1953, 2013.

XIII. APPENDIX

Join	Tables	Filters	Order By	Group By	Limit
ID					Clause
Q1	Item, Order_line	ol_o_id	None	None	None
Q2	Customer, Orders	c_uname	o_date, o_id	None	1
Q3	Customer, Address, Country	c_name	None	None	None
Q4	Author, Item	i_subject	i_title	None	50
Q5	Author, Item	i_subject	i_pub_date, i_title	None	50
Q6	Author, Item	i_id	None	None	None
Q7	Orders, Customer, Address as ship_addr, Address as bill_addr, Country as ship_co, Country as bill_co	o_id	None	None	None
Q8	Item, Shopping_cart_line	scl_sc_id	None	None	None
Q9	Item as I, Item as J	i_id	None	None	None
Q10	Author, Item, Order_line, Orders tmp table	i_subject	ol_qty	i_id	50
Q11	Order_line as ol, Order_line as ol2, Orders tmp table	ol.ol_i_id, ol2.ol_i_id <>	ol_qty	ol_i_id	5

Fig. 15. Specification of joins in the TPC-W Benchmark.

ID	Statement	ID	Statement	ID	Statement
W1	Insert Orders	W6	Insert shopping cart	W11	Update shopping cart
W2	Insert CC Xacts	W7	Insert shopping cart line	W12	Update shoppping cart line
W3	Insert Order line	W8	Delete shopping cart line	W13	Update customer
W4	Insert Customer	W9	Update Item1		
W5	Insert Address	W10	Update Item2		

Fig. 16. Specification of write statements in TPC-W Benchmark.