Informed Schema Design for Column Store-based Database Services

David Bermbach*, Steffen Müller†, Jacob Eberhardt* and Stefan Tai*
*TU Berlin, Information Systems Engineering Research Group, Email: {db,je,st}@ise.tu-berlin.de
†Karlsruhe Institute of Technology, Institute of Applied Informatics, Email: st.mueller@kit.edu

***Abstract*—While database schema options in relational database management systems are few and have been studied for decades, little effort has so far been devoted to NoSQL column stores. Today, schema design for column stores is still based on the gut feeling of the application developer instead of being approached systematically. This is even more critical as "good" schemas in column stores do not only depend on the data model of the application but also on the queries on that data: Poor schema design will either lead to a situation where not all queries can be answered or where some queries will show really poor performance. In this paper, we propose a systematic and informed approach to database schema design in NoSQL column stores by means of automated schema generation and application-specific schema ranking.**

***Keywords*—NoSQL Systems, Column Stores, Schema Design.**

I. Introduction

Enterprise application systems and service-oriented architectures typically use a dedicated database layer or database service to handle persistent storage. Apart from object databases in combination with object-oriented programming languages, though, there is always a mismatch between the internal data structure of the application and the data structure of the underlying database system. For instance, today's most common system combination – object-oriented applications running on top of relational database systems (RDBMS) – use objects and references or tables and foreign keys respectively. These data structures are not per se compatible and require some kind of mapping which has two main aspects: a *database schema* describes how the application's data is represented using the data structures of the database; *query mappers*, in contrast, are concerned with translating operations or information desires from within the application into queries that can be executed on the database and vice versa.

For RDBMS, both aspects are well studied [1]–[3]: relational database schema design is based on normalization and query mappers use object-relational mapping (ORM) techniques.

Over the last few years, though, a new class of database systems – which are typically referred to as NoSQL systems – has evolved. Usually, these systems are grouped into the three categories key-value store (e.g., Amazon S3[1]), column store (e.g., Apache Cassandra[2]), and document store (e.g., MongoDB[3]) [4]. Probably for all of these systems, but certainly so for column stores, database schema design is today mainly done based on the gut feeling of the application developer. While a systematic approach to designing column store schemas is still missing, reusing existing approaches from relational schema design will incur very poor performance:

Current column stores do not support queries spanning multiple tables and normalized schemas effectively maximize the number of tables. Therefore, a normalized schema will have the effect that a large number of queries is either not answerable or must be computed within the application. Computing queries within the application, though, does not only increase the burden of application developers but is also highly inefficient in terms of performance as well as, typically, network load. Since answering all queries is mandatory, normalized schemas will maximize the number of queries that have to be computed within the application and, thus, lead to very poor performance.

While it is, thus, certainly possible to reuse classical database schema design techniques like normalization, this will, in most scenarios, also be the most inefficient way of using column stores: The key to efficient use of NoSQL column stores lies in denormalization and data redundancy, leveraging knowledge on the concrete queries triggered by the application. Where schema options in normalization are few, though, denormalization typically leads to many different schema options which are all more or less feasible but not necessarily reasonable choices. At the moment, this schema design process is entirely left to the gut feeling of the application developer and is also a manual approach – we aim to improve this process with the following contributions of this paper:

- *A systematic approach which creates a set of feasible schema options*. This approach leverages knowledge on the application's data entities and their relationships as well as the queries on these entities.
- *A scoring function to rank the schema options*. While our evaluation demonstrates that already a very simple scoring function suffices to provide recommendations that are at least as good as the ones by column store experts, we also give an *overview of additional metrics* that could enhance the scoring function.
- *A proof-of-concept implementation which automates this schema design process*. In its current state, very few manual tasks are left which we will automate in the next prototype version.

This paper is structured as follows: In section II, we recapitulate column stores concepts as well as data denormalization techniques and discuss related work. Afterwards, in section III, we present our systematic schema design approach for NoSQL column stores, the scoring function with its potential

[1] aws.amazon.com/s3
[2] cassandra.apache.org
[3] mongodb.org

enhancements, and our proof-of-concept implementation. Next, in section IV, we discuss the current state and the limitations of our approach. Finally, we evaluate our approach by comparing its results to the recommendations of the Twissandra [5] use case in section V before concluding in section VI.

II. Background and Related Work

In this section, we will give an overview of the background relevant to our approach and related work. For this purpose, we first discuss column stores and denormalization techniques before demonstrating the differences to related approaches.

A. Column Stores

Column stores are arguably the most popular type of NoSQL system which can be seen in their widespread adoption, e.g., in the context of Apache Hadoop[4] or at Netflix[5], one of the main contributors to Apache Cassandra[6]. Furthermore, a large number of column stores is available as open source which are all more or less based on the original design proposed for Bigtable [6] – especially, the data format of Bigtable: "A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes." [6].

Basically, this design extends the simple distributed map of key-value stores so that the value of the map becomes another map which may have separate entries for any keys of the outer map. When all inner maps use the same set of keys, i.e., there is a fixed database schema, then the column store can be represented as a kind of table where the key of the outer map defines the row and the key set of the inner map(s) defines column names. While the table-like structure may look similar to database tables in RDBMS, this is deceiving as there are several key differences. For instance, the database schema, i.e., the key set of the inner map, is not fixed unless enforced within the application and only the row key is indexed so that queries which retrieve rows based on some other column have to scan the entire cluster. Recently, some systems, e.g., Cassandra, have introduced secondary indexes which are expensive in terms of performance, though. Furthermore, there is typically no support for `JOIN` operations which have to be implemented at the client if necessary. Hence, while the strength of column stores is elastic scalability, they come with a reduced set of queries they support so that neither migrating an application developed for RDBMS to column stores nor designing new applications on top of column stores can be deemed trivial.

Finally, column stores like all NoSQL systems also typically offer only eventual consistency [7], [8] with varying degrees of staleness and ordering behavior [9]–[11] and very little support for transactions in favor of availability and latency [12]. This should always be kept in mind during the schema design as any query which is broken down into multiple requests may be affected.

In this paper, we will refer to the data structure of column stores as tables, rows, and columns since we focus on the design of (fixed) database schemas based on entities and their relationships. Therefore, the flexible schema options of column stores will not have too much influence in this scenario, but talking about "columns" instead of "keys of the inner maps identical across all outer map's values" improves readability of this paper. Furthermore, while our notion of "queries" is not limited to SQL, we will in this paper use SQL-like syntax for expressing different query concepts.

B. Data Denormalization

Denormalization techniques have for many years been an active field of research in the context of RDBMS [1]–[3]. These techniques can improve performance, but come at the cost of potential anomalies and inconsistencies which require further handling. As `JOIN` operations are comparatively cheap in RDBMS, denormalization is only used to meet critical performance requirements. Typically, though, denormalization is avoided.

NoSQL systems are vastly different in that regard: `JOIN` operations are typically not supported and would be very expensive due to the distribution of data across multiple machines. Thus, denormalization becomes an essential tool in schema design – much more than a method for optimization in rare cases. Here, denormalization describes the process of reworking a database schema in a way that it is no longer in third normal form (3NF) and, thereby, trades data redundancy in favor of performance.

Based on Sanders and Shin [1], there are four denormalization strategies for RDBMS, which can also be applied to column stores: collapsing tables, splitting tables, adding redundant columns and derived attributes (i.e., results of aggregate functions).

NoSQL systems typically partition data across storage nodes automatically and derived attributes cannot be calculated on the fly by those systems as they would involve reading from the entire cluster. Collapsing tables and adding redundant columns are strategies which can help to avoid `JOIN` operations, these are, therefore, also the tools of choice in our approach (see section III-B) and related work [13]–[15]. With our approach, we do not introduce new data denormalization techniques, but rather aim to answer the question when and where to denormalize in which way.

C. Schema Selection for Column Stores

Scherzinger et al. [16] show, that schema choices in column stores can have profound effects on performance in the case of the Google App Engine[7] datastore. Obviously, optimal schema selection can not be easily derived by normalization, as for RDBMS, but rather depends on application requirements and usage patterns.

Mior [15] motivates the need for an automated schema generation approach, identifies relevant trade-offs and proposes a non-monetary cost model without specifying it in detail. No solution approach is presented, the paper merely motivates future work in the field by examining an example use case.

Vajk et al. [14] use a subset of the relational Twitter[8] data schema which they denormalize in order to identify a column

[4] hadoop.apache.org
[5] netflix.com
[6] cassandra.apache.org
[7] appengine.google.com
[8] twitter.com

store schema with minimal monetary cost. Their cost function considers storage and query cost along a predefined query load. No generalizable approach is provided, though, which would be applicable to more general use cases. Furthermore, we are not sure how reasonable it is to optimize for monetary cost only instead of considering perfomance.

Vajk et al. [13] provide a formal algorithm which derives a schema minimizing monetary cost of storage for column stores based on an initial relational schema. The proposed algorithm denormalizes by resolving functional dependencies between relations, whereas we, in contrast, will later introduce an approach leveraging knowledge on actual queries. The cost function they propose to rank schema options calculates the actual monetary cost for the service operator – the determination of an adequate cost function, however, is not described in the paper. Furthermore, the sole use of a monetary cost function implicitly assumes that there is no difference in performance between the schema options derived by the algorithm, which is usually not the case, e.g., see the work by Scherzinger et al. [16] discussed above. This approach might, therefore, overvalue inexpensive schema options with poor performance.

Furthermore, secondary indexes available in many column stores (e.g., in Cassandra or DynamoDB[9]) are not considered, which limits the value of recommendations their approach can make, as those indexes can heavily influence performance and should, therefore, also affect the schema design process. In general, we believe that the output of their approach may partly overlap with our results but that our recommendations will show much higher operational performance. Since their publication did not include an evaluation, we could not verify this hypothesis experimentally.

All these approaches seem to be in a very early stage and none of them come with an implementation enabling informed database schema design by means of automated schema generation and application-specific schema ranking.

III. A Method for Informed Database Schema Design in Column Stores

In this section, we will present our novel method for an informed schema design based on knowledge of the application's queries and data model. For this purpose, we will start with a high-level overview of our approach in section III-A, where we also discuss current assumptions and limitations. Afterwards, we will provide the details for both phases of our approach, schema creation (section III-B) and schema assessment (section III-C). Finally, we will give a brief overview of our proof-of-concept implementation which is the last necessary step towards informed database schema design for column stores leveraging automated schema generation and application-specific schema ranking (section III-D).

A. Overview and Assumptions of the Approach

The basic idea behind our approach is, to optimize the database schema for read access. The implication of this is, that each read query which is known in advance can be fulfilled by just a single request to the column store. In contrast, write queries will have to issue multiple requests per query. We made this design decision for the following three reasons:

First, optimizing for writes will lead to low numbers of requests (but not necessarily just one) per write query. This has catastrophic implications for read queries, though, which will have to issue multiple requests to the column store which leads to high data volumes transferred (as most queries then have to be computed within the application since the database system does not provide the necessary query features) as well as unpredictable quality of results in the face of eventual consistency [7], [8], [17] and lack of transactional features. In contrast, optimizing for reads will lead to just a single request per read query and low to medium numbers of requests for updates – and consistency guarantees largely do not matter. See also the detailed discussion in section IV-A. Hence, the effect of optimization is less severe for the other kind of operation when optimizing for reads.

Second, typical OLTP scenarios will involve a large proportion of read queries which makes our approach suitable for these scenarios. For scenarios without any reads (or very few reads), though, the application developer is better off not optimizing for read performance but rather for write performance, e.g., for use cases like persistence of log data. In that kind of scenario, an approach that only targets write performance, e.g., Mior [15], is a better choice.

Third, any query involving multiple requests will have to be executed within the application or an underlying middleware layer. In case of read queries, this might be the calculation of a JOIN, i.e., a cartesian product, which is rather compute-intensive and complex. In case of write queries, this would simply be issuing write requests for multiple target tables which would be neither compute-intensive nor complex for the application.

For these reasons, we optimize for reads. The optimization process itself can be broken down into a schema creation and a schema assessment phase. During the schema creation phase, all feasible schema options are created, optimizing for all known read queries. Afterwards, in the schema assessment phase, these different schema options are ranked according to a custom scoring function which may use arbitrary metrics. Figure 1 gives a high-level overview of the individual steps and phases of our approach.

The schema creation phase itself can be broken down into several steps: First, we identify all possible ways in which a single column store table could answer a given read query with just one read request; we create a table variant for each way. Second, we need to build schema options from this, including one table per read query. As some table variants might be more efficient in combination than others (which we cannot know at this step), we create the cartesian product of all table variants as schema options. Third, we try to merge tables within all schema options as far as possible since some queries (e.g., *q1* and *q3* from table I) might be subsets of each other or intersect. Fourth, we remove duplicate schema options that have been created through the merging process. Fifth, we check for all write queries whether they can still be executed. If that is not the case, (for instance, when we cannot identify all copies of the record which needs to be updated) we add additional look-up tables where necessary.

As we will see, the first two steps of the schema creation phase potentially create many different schema options which

[9] aws.amazon.com/dynamodb

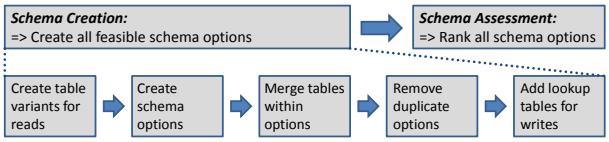

Fig. 1: Overview of the Systematic Schema Design Approach

is a both tedious and error-prone task when done manually. We, therefore, propose to fully automate our approach, e.g., by means of our proof-of-concept implementation.

Apart from the assumptions already mentioned, i.e., that we do not optimize for write-heavy workloads and that we can only optimize for queries that are known at schema creation time, our approach currently does not support subqueries[10] or aggregation functions. We believe that queries using subqueries can either be transformed into a query without subqueries or can be run sequentially by the application. Regarding aggregation functions, most column stores implement CRDT-like [18] data structures which could be used for this purpose.

B. Phase 1: Schema Creation

In this section, we will describe the different steps of the schema creation phase which vary in their respective complexity. We start with the creation of table variants in section III-B1 and continue with the creation of schema options based on the table variants in section III-B2. Afterwards, we present a set of rules according to which we merge tables within schema options (section III-B3). Finally, we describe how we remove duplicate schema options (section III-B4) and add additional look-up tables for writes where necessary (section III-B5) .

1) Step 1a: Identifying Table Variants for Read Queries: Our approach starts by analyzing all `SELECT` queries and `JOIN` queries; it identifies for each query all affected entities as well as the respective fields which are either (or both) part of the *selected fields* or the *filter fields*. Selected fields are the fields returned by a query whereas filter fields are either `JOIN` criteria or fields that are checked for meeting a certain condition (WHERE clause). For instance, *q1* from table I has the affected entity *customer*, the selected fields *fname* and *lname*, and the filter field *fname*.

Based on this, our approach builds table variants such that all executions of the respective query can be executed by a single GET request to the column store. This means that if the respective query contains a `JOIN`, then this `JOIN` will be precomputed into one single table. The table will, therefore, include fields from both entities. In either case, all selected fields will become part of the table.

To maintain uniqueness constraints on row keys, we propose, to build the row key by combining the desired filter field with the original primary key. This way, row keys are unique but we can still query the data based on the desired filter field only – in the case of Cassandra leveraging "compound keys" or for DynamoDB via "hash and range keys".

Dealing with the filter fields is more complex, since this may result in several table variants depending on the number of filter fields as well as the number of available secondary indexes. Since column stores support only row key-based look-up (and in some cases via one or more secondary indexes), selecting

[10]Subqueries are also referred to as nested queries or inner queries.

Query ID	Query
q1	`SELECT fname, lname FROM customer WHERE fname=?`
q2	`SELECT fname, lname FROM customer WHERE age=? AND address=? AND lname=?`
q3	`SELECT fname, lname, address FROM customer WHERE fname=?`
q4	`UPDATE customer SET lname=? WHERE fname=?`
q5	`INSERT INTO customer (fname,lname,age,address) VALUES (?,?,?,?)`

TABLE I: Listing of Example Queries

Row Key	Secondary Index	Other Selected Columns
age_address_lname	-	fname
age_address	lname	fname
age_lname	address	fname
address_lname	age	fname
lname	age_address	fname
address	age_lname	fname
age	address_lname	fname
-	age_address_lname	fname

TABLE II: Table Variants for *q2* with one Secondary Index

rows must either be done via the row key or a secondary index. The only alternative is reading the entire data set and filtering the records within the application which is inefficient. Therefore, all filter fields have to be stored in these columns. For instance, if a column store supports only one secondary index per table and we stick with the example from above (*fname* as single filter field), then *fname* can either be used as row key or as secondary index column. In this case, we have exactly two table variants.

While creating all table variants (even those without a row key) may seem inefficient, this is necessary as we can within this step not know which two variants of tables for different queries we might later be able to merge – thus reducing the total number of tables.

If there is more than one filter field, then we have another option: these fields can be concatenated to form a concatenated column which can be used both as row key or any of the potential secondary indexes[11]. Here, again, it is necessary that also all permutations of concatenated columns are created. As an example, table II shows all eight potential table variants, one per row, derived for query *q2* from table I in a column store with one secondary index.

This entire process is repeated for every single `SELECT` or `JOIN` query so that the output of this step is a set of table variants per query, i.e., the result is one set of table variants per read query.

2) Step 1b: Creating all Schema Options: Step 1a has produced all feasible table variants which could answer the respective query via a single GET request to the column store. Since we need a *full* database schema, though, which can answer *all* queries, we have to combine the table variants from step 1a into several distinct schema options. Again, we cannot know in this step which combinations are "good" combinations. Therefore, these schema options are calculated as the cartesian product of all table variants.

For example, if we have the two queries *q1* and *q2* from above, as well as a column store with one secondary index, then the first query has two table variants (*fname* as row key or secondary index) whereas the second query has eight table

[11]In Cassandra, such a column could later be implemented based on compound keys.

variants (see table II). Accordingly, this leads to $8 * 2 = 16$ distinct schema options.

3) Step 1c: Merging Tables within Schema Options: Each schema option which results from step 1b potentially contains many tables – one per original query. At the same time, having low numbers of tables is desirable since this reduces the overhead both in terms of storage as well as for write queries. Therefore, we try to merge tables, if possible, in this step. Note, that our approach takes each schema option and merges its tables until no further merges are possible.

For example, if we compare queries *q1* and *q3* from table I, we can see that the table of *q3* can also answer *q1* as the *q1* table is a strict subset of the *q3* table. Since the performance benefits during writes of having one table less far outweigh the downside of reading an additional field during reads[12], we can and should drop the *q1* table. The only case where this trade-off should be decided in the other direction (i.e., keeping both tables is more efficient) would be in the absence of any kind of write operation – a case where using a content delivery network would be a better solution anyhow.

This is a relatively simple case, but there are others where *dropping* a table is not an option but efficiency can still be increased by *merging* the respective tables into one. In the following, we describe five conditions which *all* have to be preserved if two source tables A and B shall be merged into a target table C. See also table III for a short overview of all conditions.

Row Key Condition: If both A and B already contain a row key, they cannot be merged[13].

Secondary Index Condition: If the number of secondary index columns, which would be required in C, exceeds the number of secondary index columns supported by the column store, then the tables can not be merged. The number of secondary indexes needed is calculated by first adding the number of distinct secondary indexes used by both tables A and B, i.e., without duplicate entries. In a second step, we decrease this number by one for every secondary index column that was built from the same filter fields as the potential key column of C.

For example, let us assume that there is only one secondary index available, table A has the secondary index column *fname_lname* and the key *id*, and table B has the secondary index column *id*. Essentially, we would require two secondary indexes (*fname_lname* and *id*) but since one of these columns (*id*) is identical to the potential key column of table C and the key column supersedes a secondary index column during merges, the merge is still possible. The result would be table C with *id* as key and *fname_lname* as secondary index.

Entity-Relationship Condition: If both Key Condition and Secondary Index Condition are met, the next condition requires checking the relationships of the original entities where the columns of A and B came from. This includes both indexed and non-indexed columns. A merge is basically (as long as the Column Count Condition – see below – is also met) possible if all columns from tables A and B are fields of either the same entity or of two or more entities that are all in direct 1:1 relationship.

If the set of original entities contains one or more 1:n or n:m relationships, a merge might in rare cases also be sensible but this requires knowledge on the actual data stored within the tables A and B. Such a merge would only be possible if both Column Count and Row Count Condition are met or if either A or B is a subset of the other. If one or both conditions are violated, the benefit of merging the tables will be outweighed by the potentially quadratic growth in data size.

Another corner case would be having at least one not even transitively related entity among the set of original entities. In that case, a merge is possible if both the intersection of the column sets from tables A and B is not empty and the column count condition is met, i.e., there is a query that has implicitly defined some kind of relationship between the respective entities. For all other cases, a merge is not possible.

Column Count Condition: If during a potential merge, the number of columns would increase for one or both of the tables by more than a certain threshold value, a merge should be avoided.

This (special) case would mean that the overhead of reading additional unutilized columns for at least one of the read queries weighs heavier than the benefit of reducing the number of tables for write operations. The threshold should be set based on the actual data stored in the table (what is the increase in terms of bytes that have to be additionally transfered over the network during reads?) as well as the expected workload (what is the ratio of reads and writes on the respective table(s)?).

In our prototypical implementation, we have for now approximated this: whenever the number of columns for at least one query increases by more than a certain configuration value (default is ten), we still merge the tables but log a warning so that the user is aware of the problem and can adapt the settings accordingly.

Row Count Condition: If we try to merge two tables A and B where A is the result of a `JOIN` on at least two entities that are not in 1:1 relationship, then the storage overhead can potentially increase in a significant way. The easiest way is never to merge tables in such a scenario, but in some cases the growth in size might be acceptable compared to the benefit of reducing the total number of tables. This would, for instance, be the case if the actually observable relationship shows low values for n and m respectively.

As a rare corner case: The same holds for merging tables A and B into a table C if C would then contain the `JOIN` of entities that are in 1:n or n:m relationship whereas A and B do not already contain columns from entities in 1:n or n:m relationship. This means, we would precompute a JOIN even though there is no JOIN query for the respective entities amongst the set of known queries. The decision whether it makes, nevertheless, sense to merge the tables depends on the actual data stored within the original entities as null values usually do not cause a storage overhead whereas additional rows do. Furthermore, denormalized 1:n or n:m relationships also increase the overhead for all write operations. Therefore,

[12]We have verified this for Cassandra in experiments.

[13]This is the reason why all table variants have to be created and why the schema options must be the cartesian product of all table variants. This will assert that all other combinations of table variants will also exist in some other schema option where a merge is then possible.

Merge Condition	Summary
Row Key	A and B may not both have a row key.
Secondary Index	Removing the row key columns from the union of the filter columns of A and B must result in a set that fits in the available secondary index columns.
Entity-Relationship	Either one table is a subset of the other, or A and B contain only fields from tables in 1:1 relationship
Column Count	For neither of the queries behind A and B may the number of columns increase by more than a threshold.
Row Count	During a merge, the number of rows may not increase by more than a threshold.

TABLE III: Overview of the Merge Conditions for Tables A and B without Corner Cases

the number of rows should not increase by more than a certain threshold.

The actual merge first builds a set of columns from the columns of both A and B. In a second step, it removes all "normal" columns for which a secondary index or row key column with the same entity-field combination exists. In the third step, it removes all secondary index columns for which a key column with the same entity-field combination exists, i.e., during merges key columns prevail over secondary index columns which in turn prevail over all other columns while duplicate columns for the same entity-field combination are removed.

4) Step 1d: Removing Duplicate Schema Options: From this step on, only the merged schema options, that resulted from step 1c, are considered. While the original schema options are all unique except for some special cases, the set of merged versions of them may contain duplicates. To increase efficiency, in this step all duplicates merged schema options are removed.

5) Step 1e: Adding Look-up Tables for Write Queries: The steps so far have only considered read queries, but there are also write queries which each have to be handled in a different way. The basic idea of this step is to check whether a particular write query can be executed based on the current set of tables or not (this will typically have a different result for any of the schema options from step 1d). In the latter case, we then add an additional table which serves as "look-up" table to identify the affected records. For this analysis, we have to distinguish queries that affect an existing data record (`UPDATE` and `DELETE` queries) and operations that create a new record (`INSERT` queries).

INSERT Queries: These queries are the easiest to handle as they do (typically) not affect existing records. For each query, we only have to identify the entities and fields updated and create a new row in all tables where one of these fields occurs. Additional fields within that table may be set to null. As an example, *q5* affects one entity (customer) and four fields (*fname*, *lname*, *age*, *address*).

There is one caveat, though: We have to check whether there is any query that updates an entity for which we also have a `JOIN` query. In that case, especially for precomputed 1:n or n:m relationships, inserting a new row requires adding either null values (if there is no corresponding instance of the other entity) or adding the correct values for the remaining columns. Potentially, inserting a value might even require to add several rows if there is more than one corresponding entity instance. While this happens at runtime and not during the schema-design phase, we still have to consider this in this phase. The only solution, we could think of, is adding a table for the data of the original *other* entity as well so that the necessary information for calculating the new rows in the `JOIN` table is available.

For obvious reasons, tables added for the purpose of `INSERT` queries may not be merged with other tables.

UPDATE and DELETE Queries: For each of these queries, we again identify the *filter fields* and *selected fields* (affected fields would be a better name here, but we will stick with selected fields). Filter fields are the fields referred to in the `WHERE` part of a query, whereas selected fields are the fields that are updated, e.g., in case of *q4* the selected field is *lname* and the filter field is *fname*.

Next, we check all tables for occurrence of the set of selected fields. Whenever a table contains a selected field, we check whether the key and secondary index columns allow a look-up via the set of filter fields of our query. If this is possible for all tables, then we are done. If not, we can choose to either add the set of filter fields as additional secondary index column or as row key (if possible) or alternatively add an additional look-up table which uses the row key and the secondary index(es) of the target table to identify the respective rows which need be updated or deleted.

These new tables may also be merged with the existing read tables as well as amongst each other. Depending on the combination of storage system, queries, and workload distribution, the use of secondary indexes might be slower than using an additional table. It is, therefore, advisable to either use both approaches in parallel and create separate schema options from them or to benchmark the performance of secondary index look-ups in advance and to just create the more efficient option.

Finally, the first approach of inserting additional secondary index columns might allow additional optimizations later-on: Apache Cassandra [19], for instance, offers so-called compound keys. These would allow us to drop the inserted secondary index column if the contained fields are a subset of the existing row key.

C. Phase 2: Schema Assessment

While the first phase of our approach has, backed by our implementation which automates this process, produced a potentially large number of different schema options, we propose to use a second phase which is dedicated to ranking the various alternatives. This is done with the aim of determining the best solution for the combination of actual column store used as well as workload details.

In this section, we describe a set of (extensible) metrics which can be combined into a scoring function (sections III-C1 to III-C3). When multiple metrics are used, these can be aggregated into a single-value score for each option based on user-specified weights for the different metrics. Building on this, we also propose a rather simple scoring function (section III-C4) which already provides a good assessment of schema options (see the evaluation in section V)

Obviously, the ranking highly depends on both the concrete system as well as the actual use case: The more information is available, the more accurate does the resulting ranking reflect the fit of the higher ranked schema options to the concrete use case. As some of that information might simply not be

available, we discuss in the following for each metric how it can be assessed in an accurate way as well as how it can be approximated.

1) Weighted Number of Operations for Write Queries: Ideally, there is for every write query, i.e., `UPDATE`, `DELETE`, or `INSERT`, just a single table which needs to be updated so that these queries can be executed by a single request to the column store. Since our approach optimizes for read queries, this will rarely be the case. In typical cases, `UPDATE` and `DELETE` queries will have to issue a read request to a look-up table before actually being able to issue the update or delete requests for all the tables with redundant copies of the respective data set. `INSERT` queries may have to read a table if they affect entities that are part of a `JOIN` query.

This, obviously, hurts performance a lot as every request also adds a network round trip to the query latency. This metric, therefore, calculates for every query the number of actually resulting requests to the database system in a first step. Since some queries might be issued more frequently than others, the second step then calculates the weighted average of these numbers, using the frequency of the respective query as weight.

If the frequency of different write queries is not known[14], an approximation may just calculate the average instead of the weighted average. Alternatively, an even rougher approximation could simply use the total number of tables.

2) Storage Overhead: When using a cloud-hosted column store service like Amazon DynamoDB, the storage overhead of storing redundant copies of some entities' fields directly translates into monetary cost. Even when running an open source column store, e.g., Cassandra [19] or HBase[15], the storage overhead for very large clusters can quickly become a problem. This is especially true for Cassandra, for which [20] have shown that the cluster performance significantly degradates when confronted with very large data volumes per machine. This might necessitate spawning additional instances to handle the amount of storage with acceptable performance.

For these reasons, it is important to select a schema option where the storage overhead, compared to the memory requirements of a normalized database schema, is as low as possible. In our storage overhead metric, we have to consider two dimensions: The first dimension is caused by having redundant copies of columns in different tables which is almost guaranteed to be the case for at least a few columns in every schema option. The second dimension is caused by precomputing JOINs, i.e., by denormalizing 1:n or n:m relationships of entities into a single table which typically causes additional rows in that table in comparison to the normalized data schema.

The first dimension can be measured relatively straightforward by counting for each entities' fields how often they are stored redundantly. Again, we can calculate the weighted average of these values by using the average data size of the respective field as weight. The resulting number is the expected factor by which the data volume will grow when using that schema option compared to the bare storage needs of an entirely normalized schema. When the average data size of the fields is not known, which will at least for application migration scenarios never be the case, an approximation would be to assume identical sizes for all fields.

The second dimension is more difficult to measure: Apart from knowledge on the number of instances of each entity A and B (which will typically change at runtime anyhow) as well as average field sizes, it also requires information on the degree of matching of the joined fields. Depending on the actual degree of matching the second dimension may cause any overhead in the range of zero to n times the number of instances of entity A plus m times the number of instances of entity B. The degree of matching may be known for cloud migration scenarios but rarely for any other application scenarios. Therefore, it is really hard to calculate the expected storage overhead of the second dimension. In general our approach will only then denormalize 1:n or n:m relationships if there is a `JOIN` query spanning both entities. In that specific case, though, this overhead will be identical for all schema options. Since our metric does not calculate cardinal values but rather orders schema options on an ordinal scale, the resulting order of schema options cannot be affected by this second dimension. It is, hence, safe to ignore the second dimension of the storage overhead metric.

3) Datastore-Specific Metrics: While the creation of schema options is largely independent from the actual column store, the performance of different operations (e.g., look-up via secondary indexes) may vary between different column stores. Therefore, we propose to also benchmark several performance numbers of the column store and use these to influence the ranking of schema options.

First, if the column store supports secondary indexes, it is necessary to measure the performance differences of retrieving rows based on their row key or the different secondary index columns. Using this information, we can then calculate an expected row retrieval performance for each query. Using the frequency of each query as weights, this enables us to calculate the weighted average of these performance numbers which will be the expected read performance for read queries.

The actual performance may, of course, highly depend on the size of the rows retrieved as well as the number of rows in the table (e.g., in case of Cassandra the number of items within one column family) and may also vary over time. Still, it is highly unlikely that the order of these look-up options (via row key or secondary indexes) varies between benchmark and actual deployment later on. This implies that the ordering of schema options will almost always be correct, i.e., while the cardinal values may deviate, the ordinal scale will not be violated. Obviously, the accuracy of the cardinal values depends on the similarity between benchmark and actual deployment. As an approximation, we can again use weights of one if the frequency of queries is not known.

Second, we can benchmark the performance overhead of reading an additional surplus column as a function of the additional data size transferred over the network. Based on this, we propose to calculate the performance overhead for each read query which results from reading additional columns. Using the frequency of the respective queries as weights again, this can be used to calculate the weighted average performance overhead of read queries. Obviously schema options where less merges were possible in step 1c will rank better in this

[14]This information will rarely be available when developing a new application but will almost always be available when migrating existing applications.

[15]hbase.apache.org

dimension. As an approximation, we propose to replace this metric by increasing the weight for storage overhead in the overall scoring function instead.

4) A Simple Scoring Function: For our evaluation, we have used a very simple scoring function as a heuristic which already provides good recommendations (see section V). The discussed metrics are likely to lead to the optimal schema but involve a large information gathering and benchmarking effort, the benefit of which might simply not be worth the effort. This scoring function builds on two metrics:

Average number of secondary indexes used: This metric is defined as the total number of secondary index columns, that a given schema option uses, divided by the number of tables. For this metric, lower values are better since row key-based access is much faster in Cassandra than secondary index-based look-ups[16]. For other systems, this may be different but is unlikely to be so.

Average data duplication: This metric is defined as the total number of columns of a schema option divided by the total number of fields of the original entities. For this metric, lower values are better since they imply a lower storage overhead as well as fewer tables, i.e., less requests for write queries.

We have used the sum of both as a scoring function, i.e., schema options with a higher score are considered better. Alternative scoring functions and metrics may lead to identical or entirely different rankings – what is desirable, depends on the use case and the preferences of the application developer.

D. Implementation

We have prototypically implemented our approach in Java. For the schema creation phase, users have to specify the entities, their fields, and relationships as well as the queries which shall be considered. For the schema assessment phase, users can specify to reuse any of the existing scoring functions. Alternatively, they can subclass an abstract class to create arbitrary scoring functions.

In its current version, our prototype fully automates all steps apart from step 1e (see section III-B5) which is not too difficult to do manually for the moment. We plan to publish the proof-of-concept implementation as open source upon completion of the implementation.

As a sidenote, we are also currently working on the benchmarking middleware proposed in [21] for which we plan to use this implementation within the mapper module.

IV. Discussion

In this section, we discuss the impact of eventual consistency as well as poor transactional guarantees on database schemas designed either via our approach or the approaches of Mior [15] and Vajk et al. [13], [14]. We also discuss limitations of our approach.

A. Cost of Inconsistency and Poor Transactional Features

To our knowledge, there is no publicly available column store that actually guarantees strict consistency or multi-key transactions[17]. This means that every query, that is – during execution – broken down into several requests to the storage system, will be affected by the lack of transactional guarantees and arising inconsistencies. In this section, we try to shed some light on both influence factors.

The lack of multi-key transactions foremost means that the application (or a middleware layer) is responsible for asserting atomicity of all queries that require more than one request to the datastore. If necessary, partial requests have to be retried until they succeed as a rollback will typically be not feasible. Furthermore, in the presence of concurrent write requests, any query comprising more than one request may see an inconsistent snapshot of the database due to missing isolation guarantees.

Basically, in read-optimized schemas, there are three types of queries which may be broken down into multiple requests: `UPDATE` or `DELETE` which use a look-up table to identify the respective target records, `INSERT` queries which must update several tables, and `SELECT` queries (potentially including a `JOIN`) which read from more than one table.

`UPDATE` and `DELETE` queries may leave inconsistent data if there are several concurrent requests to the same table. If all identical queries with varying parameters are composed of sub-requests that are executed in the same order, this is highly unlikely to occur. In either case, it is desirable to have as few tables as possible to reduce this risk. There is a high probability that the conflict resolution mechanisms of the underlying datastore can resolve the remaining inconsistencies as long as the execution order of sub-requests is preserved.

Our approach cannot avoid these issues as we primarily optimize for read performance. A scoring function emphasizing low total numbers of tables, though, will help to select a schema where this issue is unlikely to occur. In contrast, the approach of Mior will not encounter any inconsistencies due to concurrent executions of `UPDATE` and `DELETE` queries and the approach by Vajk et al. seem to be affected more than ours as it will result in a higher number of tables.

`INSERT` queries can only cause inconsistencies when their sub-requests are not fully executed. So, as long as atomicity is asserted via retries on the application-side, inconsistencies are highly unlikely to occur in our approach or the one by Vajk et al. Again, the approach of Mior will not encounter any inconsistencies for these queries.

`SELECT` and `JOIN` queries which read from more than one table at a time to aggregate the individual requests' results into a query result may show catastrophic consistency behavior. First, these operations are bound to discover any inconsistencies caused by write operations which is unavoidable. Second, aggregated read results based on several requests will frequently show dirty reads in the presence of writes. The main issue here is that they may not only return stale data (which in itself may be acceptable [22], [23]) but may also combine different version snapshots of related entities. For instance, if a query updates two tables in a certain order in parallel while we read both tables in a different order, results may not only be stale but entirely meaningless. The eventually consistent [7], [8] guarantees of today's column stores further aggravate this

[16]Based on extensive experimentation with Cassandra beforehand, we had noticed that look-up via secondary indexes is by dimensions slower than row key-based look-up.

[17]The Google App Engine datastore supports partial transactions within small groups of rows, the so-called entity groups.

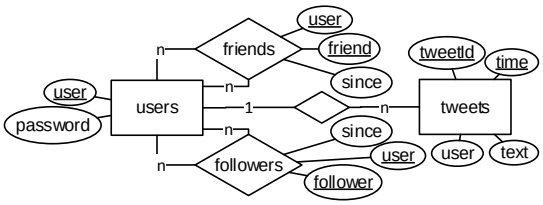

Fig. 2: Normalized Data Model of Twissandra

problem.

While our approach may encounter inconsistencies caused by multi-request write queries, it is safe from encountering any issues stemming from aggregating results of several distinct read requests. The approaches by Mior or Vajk et al. can rarely return usable results in read queries of realistic application workloads since they are bound to read multiple tables.

In summary, we believe that our approach has the best chance of dealing with inconsistencies in an appropriate way.

B. Limitations of our Approach

While our contributions offers important steps for an informed database schema design for column stores and the automation of that process, we can still identify two issues:

First, applications may change over time: New queries may be added, existing ones may be changed or removed, the number of entities and their respective fields may change, etc. As our approach can only optimize for known queries, this may lead to situations where our recommended schema is no longer a good schema. Still, the flexible schema options of column stores allow us to easily adapt the schema at a later time.

Second, column stores seem to resemble each other closely – but only at first sight. They actually differ in crucial areas which may affect the schema design: E.g., if the store supports list columns, it may be more efficient to use two tables with list columns instead of denormalizing an n:m relationship. Also, column stores support different degrees of complexity in `WHERE` clauses. Our approach currently assumes that the data store supports the respective operator and that conditions are only connected by logical ANDs. Logical ORs can be executed by breaking the query into a sequence of queries. A general solution to this problem is beyond the scope of this work.

V. Evaluation

In this section, we show that our method and its prototypical implementation produce correct results and are applicable to realistic application scenarios. For this purpose, we use Twissandra [5] – a sample application offering *Twi*tter-like functionality on top of Apache C*assandra* – as use case. Twissandra was also used as a use case by Vajk et al. [14] and has been developed by the Cassandra community as a reference implementation for the features of Cassandra. Since this community can be deemed experts for column stores, this scenario shall, therefore, illustrate that our automated approach recommends solutions at least as good as a schema manually developed by column store experts.

We start by describing the normalized data model which Twissandra is based on. Next, we quickly recapitulate the denormalization and its optimization described by Todorov [5]. Finally, we describe how we parameterized our proof-of-concept

Query ID	Query
t1	`SELECT * FROM tweets WHERE tweet_id=?`
t2	`SELECT tweets.time, tweets.tweet_id FROM users, tweets WHERE users.user=? AND users.user = tweets.user`
t3	`SELECT tweets.time, tweets.tweet_id FROM users, tweets WHERE users.user=? AND users.user = tweets.user AND tweets.time=?`
t4	`SELECT * FROM users WHERE user=?`
t5	`SELECT friends.friend FROM friends, users WHERE users.user=? AND users.user = friends.user`
t6	`SELECT followers.follower FROM followers, users WHERE users.user=? AND users.user = followers.user`

TABLE IV: Listing of Read Queries in Twissandra

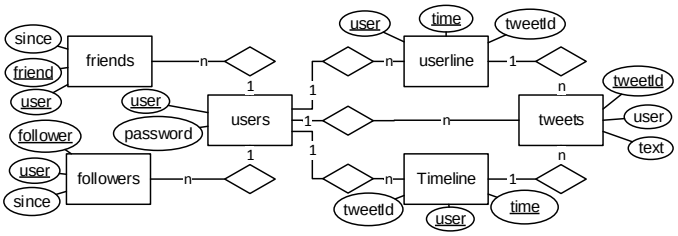

Fig. 3: Data Model of Twissandra

implementation which – at first try – recommended a schema exactly identical to the solution developed by the Cassandra community.

A. Normalized Data Model and Queries

Twissandra has four entities (see also figure 2): users, tweets, friends, and followers. Each user can have several friends and followers; he may also have posted an arbitrary number of tweets. Table IV provides an overview of all read queries: There are queries for selecting a tweet based on its ID (*t1*) as well as for identifying tweet IDs based on either the author (*t2*) or on the author and the respective time (*t3*). Other queries retrieve user data based on the username (*t4*) or return follower (*t6*) and friend (*t5*) data for a given user. Finally, there are write queries which create or delete friends and followers, create new users, or create a new tweet. These queries were identified based on the publicly available implementation of Twissandra[18].

B. Denormalization and Optimization

Todorov [5] describes that they partially denormalized this data model in a first step to accommodate the two `JOIN` queries which select tweet IDs based on either user (*t2*) or both user and time (*t3*). For each of them, an additional table is created in Cassandra: The userline which holds references to all queries a particular user has posted and the timeline which holds references to all tweets one's followees have posted. Since the tweet ID is a very small value, this introduces a negligible storage overhead. See figure 3 for an overview of the proposed schema.

Since every query to the userline or the timeline table is inevitably followed by a query to retrieve the respective tweet text (*t1*), it is possible to inline the tweets table into both userline and timeline tables. This comes at a relatively high cost in terms of storage overhead but approximately cuts read latency for userline and timeline in half. This optimization is shown in figure 4.

[18] github.com/twissandra/twissandra

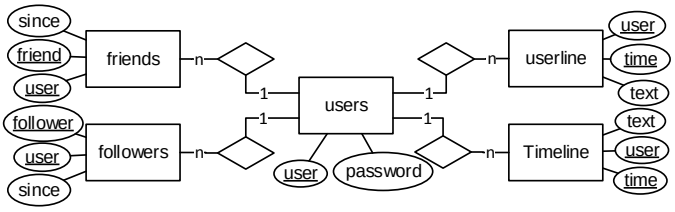

Fig. 4: Optimized Data Model of Twissandra

C. Comparison to our Approach

To evaluate our approach, we next parameterized our proof-of-concept implementation with the normalized data model and the original queries. We used the scoring function described in section III-C4 mainly to demonstrate that already very little information can be used to guarantee "good" recommendations.

The output of our tool were 128 total schema options with the schema from figure 3 as the option with the highest score. Originally, we had planned to run a system benchmark comparing both the Twissandra schema and our recommended schema and had even implemented a benchmarking tool for this very purpose. Since our recommendation was identical to the option chosen by Twissandra, we skipped these additional experiments. Nevertheless, we would recommend to benchmark at least the top five or top ten recommendations in practice.

As discussed above, there is the potential optimization of inlining the tweets table into both the userline as well as the timeline table. This is a good example where our assumption of knowing all important queries a priori matters: When we change the queries *t2* and *t3* so that they return "`*` `FROM` *tweets*" instead of "*tweet_id* `FROM` *tweets*" and remove the (no longer needed) query *t1* which retrieves tweets by their ID, then our approach also recommends the optimized schema shown in figure 4. We verified this with our prototypical implementation where we made no further changes beyond these three affected queries.

VI. Conclusion

Database schema design for RDBMS has been studied for decades, for the much newer NoSQL column stores, though, there is little work yet. Hence, application developers are more or less left with their gut feeling. This problem does not only occur during the development of new applications but also whenever an existing RDBMS-based application is migrated to a column store-based cloud service. To address this issue, we have in this paper presented a systematic approach for informed database schema design in column stores via automated schema creation and application-specific schema ranking.

During the schema creation phase, our approach first creates all feasible schema options which could answer a given set of read queries with just a single read request each. Next, these schema options are optimized by removing or merging surplus tables according to a set of specified conditions. Finally, in the ranking phase, the schema options are rated and ordered using a custom scoring function. For this purpose, we have developed and presented a set of metrics as well as a simple scoring function. Finally, we have discussed how our approach as well as alternatives from related work fare in the face of inconsistencies and little or no transactional guarantees and pointed out limitations of our approach.

We have implemented the schema creation and assessment process to automate our approach. Then, to evaluate our method, we have compared the Twissandra use case, which was developed by proven column store experts, to the schema our implementation recommends for the same use case; the result was that our recommendation is identical to the one developed in the Twissandra use case.

References

[1] G. L. Sanders and S. Shin, "Denormalization effects on performance of rdbms," in *HICSS 2001*. IEEE, 2001.

[2] S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses," *Decision Support Systems*, 2006.

[3] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. van Steen, "Service-oriented data denormalization for scalable web applications," in *WWW 2008*. ACM, 2008, pp. 267–276.

[4] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A Survey of Large Scale Data Management Approaches in Cloud Environments," *IEEE Communications Surveys and Tutorials*, vol. 13, 2011.

[5] L. Todorov, "Python driver overview using twissandra," *http://planetcassandra.org/blog/python-driver-overview-using-twissandra (accessed Sept 1, 2014)*, 2014.

[6] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM TOCS*, vol. 26, no. 2, pp. 1–26, 2008.

[7] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems: An overview of models, metrics and measurement approaches," in *NETYS 2013*. Springer, 2013.

[8] W. Vogels, "Eventually consistent," *Queue*, vol. 6, October 2008.

[9] D. Bermbach, S. Sakr, and L. Zhao, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *TPCTC 2013*. Springer, 2013.

[10] D. Bermbach. and S. Tai, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," in *IC2E 2014*, 2014.

[11] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade offs in commercial cloud storages: the consumers' perspective," in *CIDR 2011*, vol. 11, 2011.

[12] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, 2012.

[13] T. Vajk, P. Feher, K. Fekete, and H. Charaf, "Denormalizing data into schema-free databases," in *CogInfoCom 2013*. IEEE, 2013, pp. 747–752.

[14] T. Vajk, L. Deák, K. Fekete, and G. Mezei, "Automatic nosql schema development: A case study," in *PDCN 2013*, vol. 2013.

[15] M. J. Mior, "Automated schema design for nosql databases," in *ACM SIGMOD 2014 PhD Symposium*, 2014.

[16] S. Scherzinger, E. C. De Almeida, F. Ickert, and M. D. Del Fabro, "On the necessity of model checking nosql database schemas when building saas applications," in *TTC 2013*. ACM, 2013, pp. 1–6.

[17] A. S. Tanenbaum and M. V. Steen, *Distributed Systems - Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2007.

[18] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *SSS 2011*. Springer, 2011, pp. 386–400.

[19] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS OSR*, vol. 44, no. 2, pp. 35–40, 2010.

[20] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," *PVLDB*, vol. 7, no. 12, 2014.

[21] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar, "Towards an extensible middleware for database benchmarking," in *TPCTC 2014*. Springer, 2014.

[22] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *ACM SOSP 2013*.

[23] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai, "A middleware guaranteeing client-centric consistency on top of eventually consistent datastores," in *IC2E 2013*. IEEE, 2013.