Automated Schema Design for NoSQL Databases

Michael J. Mior [*] [†]
University of Waterloo
mmior@uwaterloo.ca

ABSTRACT

Selecting appropriate indices and materialized views is critical for high performance in relational databases. By example, we show that the problem of schema optimization is also highly relevant for NoSQL databases. We explore the problem of schema design in NoSQL databases with a goal of optimizing query performance while minimizing storage overhead. Our suggested approach uses the cost of executing a given workload for a given schema to guide the mapping from the application data model to a physical schema. We propose a cost-driven approach for optimization and discuss its usefulness as part of an automated schema design tool.

Categories and Subject Descriptors: H.2.2 Database Management: Physical Design
Keywords: NoSQL; schema optimization; workload modeling

1. INTRODUCTION

The problem of optimizing database schemata to improve performance is well-studied in relational databases. However, NoSQL databases (not based on tabular relations) often lack a formal schema specification understood by the database engine. The data model provided by the database engine is typically primitive structures with higher-level abstractions managed within the application. Since the database engine cannot manipulate or understand this higher-level data model, it is important for the application to make efficient use of the available primitives. However, no standard tools exist for schema optimization in NoSQL databases.

1.1 Relational Schema Optimization

In general, tools for dealing with relational database schemata are far more developed than tools available for NoSQL databases. Many of these tools currently exist in commercial products. Microsoft AutoAdmin[2] is a tuning wizard which suggests materialized views and indices for Microsoft SQL Server.

Another example is the DB2 Design Advisor[17] which can recommend physical designs, partitioning, and materialized views for a given workload. The major challenge for the DB2 Design Advisor was the large search space produced by interactions between various features. They were able to achieve almost 100% performance improvement within a bounded time.

Oracle 10g introduced a feature known as Automatic SQL Tuning[3]. Tuning is a three step process: identifying high load queries, finding ways to improve execution plans, and implementing corrective actions. Of relevance to our work is the creation or removal of "data access structures" such as indices or materialized views to improve query performance. The optimizations performed via tuning were shown to significantly outperform even manual tuning by experienced database administrators.

All of these tools rely on input from the database system's query optimizer. In many NoSQL systems, the simple data model and query language means that an optimizer for higher level queries does not exist. This increases the importance of effective schema design since there are few opportunities for optimization given a fixed schema.

Vertica[10], consists of a SQL interface built on the C-Store column-oriented database[14]. It uses multiple encoding techniques to efficiently store denormalized views to improve query performance. Earlier work by Rasin and Zodnik[11] also discusses automated schema design for C-Store. However, these techniques cannot be easily generalized to apply to NoSQL database architectures.

1.2 NoSQL Schema Design

Scherzinger[13] identifies the need for good schema design in NoSQL stores by identifying the high cost of poor schema design. They show that a naive schema design results in 20–35% of write transactions failing for a given workload. This is a result of trading off low write throughput for high read throughput and optimistic concurrency control. The problem is completely avoided by selecting a better schema.

NoSQL databases present several new issues which are not solved by relational schema design tools. There is not a clear separation of the application data model and physical schema in a NoSQL database. Mappings between these the application data model and the physical schema are typically ill-defined and dependent on knowledge of the database administrator. We claim that while obvious mappings may exist for some applications, these mappings may not be optimal.

[*]Expected graduation date: August 2017
[†]Supervised by Kenneth Salem

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06.
http://dx.doi.org/10.1145/2602622.2602624.

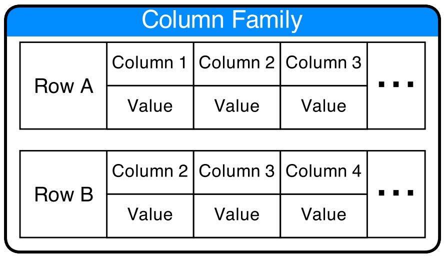

Figure 1: Data layout in wide column stores

In addition, applications are dependent on the particular physical schema as denormalization cannot be managed at the database layer. Applications are also required to implement more complex queries on top of the primitive operations provided by the database. This generally results in significant denormalization in the physical schema to allow efficient responses to queries. The database has no knowledge of the applications higher level queries and cannot directly aid in optimization.

We are aware of one existing tool, by Vajk et al.[16], which attempts to solve some of these problems. Their system starts with a normalized schema and produces an optimized schema based on the cost of executing a given set of queries. However, their model of query costs is not very expressive and has not been thoroughly validated. They do however identify a tradeoff between query execution time and the storage cost of denormalization which is relevant to our approach. SimpleSQL[6] provides a relational layer on top of Amazon SimpleDB which automatically creates the necessary schema to answer queries. However, SimpleSQL makes specific assumptions about the data model and consistency properties of the underlying datastore. This makes the approach difficult to generalize for other datastores.

Cattell[7] provides a convenient taxonomy of NoSQL databases. The three main categories discussed are document stores, key-value stores, and wide column stores. Document stores treat the data as individual "documents" which have application-defined properties and typically allow arbitrary nesting of properties. A key-value store maps data which is mostly unstructured and often opaque to simple keys. This architecture resembles a distributed hash table. Finally, wide column stores have a two-tiered structure where rows are mapped by keys to a set of columns which map to values. For simple namespacing, rows can be grouped together under a "column family" (somewhat analogous to tables in relational databases). A simple illustration is given in Figure 1. Our initial target is wide column stores such as Cassandra[9] and HBase[1], but we intend for our approach to be general enough to generate schema for each of these types of system

2. MOTIVATION

2.1 Storage Tradeoffs

In relational databases, it is common to use materialized views to provide fast answers to common queries. This approach is also used in NoSQL databases, with the views maintained by the application instead of the database. Having all the necessary data to answer a query already prepared increases performance However, as the number of materialized views increases, the storage used for the views also grows. It is impractical to maintain materialized views for all possible queries, as this would result in unbounded storage usage. A database administrator must decide which views to materialize for a given storage budget. However, we argue that there are too many alternatives to be efficiently evaluated by a human operator. We present an example that further describes some of these tradeoffs below.

2.2 View Maintenance

In addition to storage overhead, denormalized data also requires updates to maintain consistency. This overhead depends on the frequency of updates to the data and the amount of denormalization. The choice of schema also affects the cost of updates to an individual piece of data. To perform an update, the application must find the data which needs to updated. If the data is significantly denormalized, this may require a scan of large volumes of data, or further views to support efficient updates.

2.3 Transactions

Many NoSQL databases have limited or no support for transactions. For example, Cassandra provides very limited support for transactions, consisting of compare-and-set operations on single rows. However, data which is in the same row can be updated atomically. If strong consistency guarantees are required, then it may be necessary to collocate data and sacrifice opportunities for denormalization.

3. EXAMPLE

As mentioned above, applications using a NoSQL database often have a significantly denormalized physical schema. While this allows for more efficient query processing[12], it comes with the tradeoff of additional storage for denormalized data. Below we describe a realistic scenario where there are multiple choices of physical schemata, each with different space-time tradeoffs.

Consider a possible data model for a hotel reservation system given by [8] and shown in Figure 2. Suppose we have the following queries:

- Q1. For a given guest, return the hotels that guest has stayed at.
- Q2. For a given guest, return the amenities that have been in rooms the guest has stayed at.
- Q3. For a given guest, return the points of interest near hotels the guest has stayed at.

3.1 Physical Schema Options

If we consider the relations traversed in these three queries, as shown in Figure 3, we see that all three have a common prefix of Guest $\rightarrow$ Reservation $\rightarrow$ Room. We may therefore conclude that it would be useful to maintain a data structure mapping guests to rooms. For illustrative purposes, we express the data structures as column families in a wide column store (as in Figure 1). To fully answer the queries, we could use this structure, followed by a lookup of the remaining entities in the chain. This incurs a lower space penalty than structures providing direct answers to each query, while being more efficient than scanning each entity.

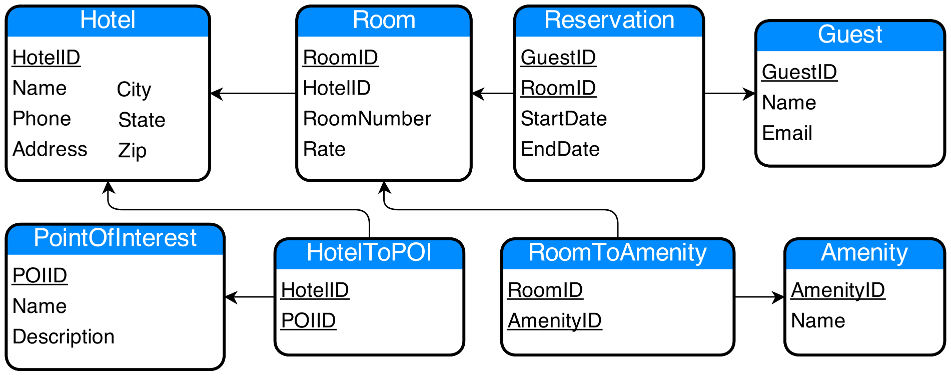

Figure 2: Example entities for a hotel booking system

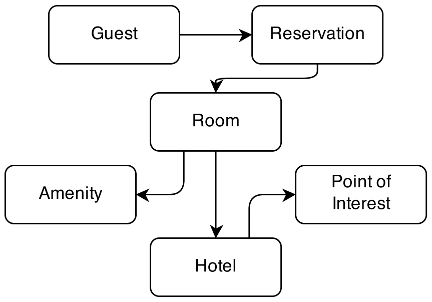

Figure 3: Query paths on hotel entities

To explore the tradeoffs, we consider 3 possible schemata:

1. A trivial mapping analogous to relational databases for each structure in Figure 2 (i.e. the primary key is the row key which maps to columns and values for each entity)
2. Materialized views for each query which map the primary key of the Guest to the relevant data
3. A materialized view which maps guest primary keys to room primary keys, followed by materialized views which can answer the remainder of the query
   (a) Room primary keys to hotel data
   (b) Room primary keys to amenity data
   (c) Room primary keys to point of interest data

We note that the second two schemata do not contain the same information as the first schema (notably, reservation information is discarded). The structures in the first schema must be included in the final design to avoid losing data and the ability to answer other queries. This consideration must be taken into account by an automated tool, but we note that in a complete workload, we would expect all data to be accessed. To estimate query and storage costs for each of these possible schemata, we consider the following data:

- 1000 hotels (n_H)
- An average of 25 points of interest per hotel ($n_{P,H}$)
- An average of 250 rooms per hotel ($n_{R,H}$)
- An average of 5 amenities per room ($n_{A,R}$)
- 100,000 guests (n_G, average rooms per guest is 5)
- An average of 1000 reservations per hotel (each for random room with a random guest) ($n_{R',H}$)

While this is not true in practice, we assume that all records stored have uniform size and retrieval costs since this does not affect our argument. We further assume that all guests have a unique hotel and room for each reservation. With this information, we can calculate the size of the database for each schema, which is given in Table 1.

Table 1 also gives the mappings which are present in each schema. We use these mappings to construct query plans for each schema. We also count the number of primitive operations required to execute the queries under each plan.

A primitive operation is one retrieval of either an item identifier or item data. Multiple items retrieved from a single row are counted as unique operations. The application responds to queries by fetching data from the data structures defined in Table 1. Data is fetched by keys which are either inputs to the query (in the case of guest IDs) or IDs which are found as the result of other operations on the database. We note that in a wide column store, we can map row keys to a list of IDs by using the IDs as column names with no data. The plan for each query in the first schema is given below.

Plan for Schema 1

Q1. (a) Get room IDs for the given guest ID
5 operations on $s_{1,8}$ (5 rooms/guest)
(b) Get all hotel IDs from the room IDs
5 operations on $s_{1,5}$ (1 hotel/room)
(c) Get hotel data from the hotel IDs
5 operations on $s_{1,1}$

Q2. (a) Get room IDs for the given guest ID
5 operations on $s_{1,8}$ (5 rooms/guest)
(b) Get all hotel IDs from the room IDs
5 operations on $s_{1,5}$ (1 hotel/room)
(c) Get all POIs from the hotel IDs
$5 \times 25 =$ **125 operations** on $s_{1,2}$ (25 POIs/hotel)
(d) Get POI data from the POI IDs
$5 \times 25 =$ **125 operations** on $s_{1,3}$ (25 POIs/hotel)

Schema 1		
$s_{1,1}$	HotelID → Hotel	$n_H = 1,000$
$s_{1,2}$	HotelID → POIID	$n_H \times n_{P,H} = 25,000$
$s_{1,3}$	POIID → POI	$n_H \times n_{P,H} = 25,000$
$s_{1,4}$	HotelID → RoomID	$n_H = 1,000$
$s_{1,5}$	RoomID → Room	$n_H \times n_{R,H} = 250,000$
$s_{1,6}$	RoomID → AmenityID	$n_H \times n_{R,H} \times n_{A,R} = 1,250,000$
$s_{1,7}$	AmenityID → Amenity	$n_H \times n_{R,H} \times n_{A,R} = 1,250,000$
$s_{1,8}$	GuestID → Reservation	$n_H \times n_{R',H} = 1,000,000$
	Total	**3,802,000**
Schema 2		
	GuestID → Hotel	$n_H \times n_{R',H} = 500,000$
	GuestID → Amenity	$n_{A,R} \times n_H \times n_{R\prime,H} = 5,000,000$
	GuestID → POI	$n_{P,H} \times n_H \times n_{R\prime,H} = 25,000,000$
	Total	**30,500,000**
Schema 3		
	GuestID → RoomID	$n_H \times n_{R',H} = 1,000,000$
	RoomID → Hotel	$n_H \times n_{R,H} = 250,000$
	RoomID → POI	$n_H \times n_{R,H} \times n_{P,H} = 6,250,000$
	RoomID → Amenity	$n_H \times n_{R,H} \times n_{A,R} = 1,250,000$
	Total	**8,750,000**

Table 1: Data size for sample schemata

Q3. (a) Get room IDs for the given guest ID
5 operations on $s_{1,8}$ (5 room/guest)

(b) Get all hotel IDs from the room IDs
5 operations on $s_{1,5}$ (1 hotel/room)

(c) Get all amenity IDs from the room IDs
$5 \times 5 =$ **25 operations** on $s_{1,6}$ (5 amenities/room)

(d) Get amenity data from the amenity IDs
25 operations on $s_{1,7}$

Adding up the operations for each query in the plan gives a total of 335 operations required to execute the first schema. We can construct similar plans for the queries in schemata 2 and 3 and see that they require 155 and 170 operations respectively. We can conclude that the views constructed for schemata 2 and 3 are able to answer the same queries using approximately half the number of operations. Therefore queries over schemata 2 and 3 are more efficient at the cost of additional storage for the denormalized data.

However, we note that the views in schema 2 occupy over three times the space of the views for schema 3. While we do not model update cost explicitly, schemata 2 and 3 incur significant costs for updates to Hotel, POI, and Amenity data. This is because this data is replicated for each guest, a scan is required to find the necessary data to update. In a real workload including updates, we could construct additional views to support efficient updates.

For more complicated examples than the one given above, the potential tradeoffs are too numerous to be evaluated by a human operator. As discussed in [2], the number of materialized views which can support a given query in a relational database is quite large. We note that there are a number of other views which would support our example workload and offer different space-time tradeoffs. Furthermore, as more queries are considered, the possibility of combining different views to save storage increases. This results in a explosion of possible materialized views for NoSQL databases. This suggests that an automated method of selecting physical schemata is as relevant to NoSQL databases as it is to relational databases.

4. MODELING

4.1 Workload Modeling

To evaluate the approach discussed above for efficient query evaluation while minimizing storage space, we require a cost model for queries. Developing this model necessitates both a language for expressing queries and a means of describing possible plans for executing the query. Benzaken et al.[4] propose a calculus for NoSQL databases, but it is overly complex for our purposes. Jaql[5] provides a generic query language over unstructured data, but is closely tied to the physical data model. In contrast, GMAP [15] proposes a query language similar to SQL which is independent of the physical data model. We adopt a similar approach using a simple grammar based on queries over entity-relationship (ER) diagrams with the basic grammar identified below.

```
SELECT [attributes] FROM [entity] WHERE [column]
(=|<|<=|>|>=) [value] AND ... ORDER BY [columns]
```

To increase simplicity, we allow attribute selection from only a single entity. Relaxing this restriction is tantamount to allowing joins, which are generally not supported by NoSQL databases. We expect to be able to relax this restriction by creating rich materialized views. However, we do use attributes from other entities in selection predicates and ordering clauses, with a simple restriction. The target attribute must be retrievable via a one-way relationship from the entity we use for selection.

4.2 Physical Design Space

To support the selection predicates and ordering clauses described above, we propose a DBMS-independent method of constructing indices and materialized views to be able to answer these queries. Along with a DBMS-specific cost model, we require a mapping from application data model (as in Figure 2) to a physical schema. This mapping is DBMS-specific to allow this approach to adapt to different NoSQL databases. We currently consider only wide column stores and note that our model is viable in both Cassandra and HBase. We first consider a query which only specifies selection predicates and no ordering clauses. To construct an index for this query, we construct a column family which will serve as the index. The row keys in the column family correspond to values for the index, and column names correspond to keys for the rows being retrieved.

For example, consider an index for the simple query `SELECT foo FROM bar WHERE baz=3`. We simply use the indexed values as row keys in a new index column family. The column names are the keys for the target entity. Note that this same index can be used to answer range queries by selecting rows within a given range. We support multiple selection predicates by concatenating column values into the row key. Currently, this model only allows range queries over a single column, which would simply be placed last in the concatenation.

In order to support ordering clauses, we use a similar index structure, but leverage the column names. Instead of placing the row key in the column name, we use the value being used for ordering with the row key as the column value. We can support ordering over multiple columns by concatenating values into the column name. We do not currently support range queries with ordering clauses.

4.3 Cost Modeling

The cost model is a component which is intended to be replaced to target a new type of system. Given the index structures discussed above, the cost model must be able to evaluate the cost of retrieving data from these structures. This is guided by estimates on the cardinality of each field and the number of each type of entity being modeled.

Therefore, we have the following inputs to the model:

1. Types of entities and indices with their fields

2. Estimated number of each type of entity

3. Estimated cardinality of each field

With this input, the cost model should be capable of estimating the cost of retrieving a given list of fields for a set number of records on an index. To validate this design, we examined in-memory read-only workloads on wide column stores. Knowledge of these systems and experimental validation has shown that a simple linear model accurately predicts the cost of simple queries in both Cassandra and HBase. We do not present the details here, but state that the cost of executing queries has a linear relationship with the amount of data fetched.

5. FUTURE WORK

The primary goal of future work will be to construct a tool which solves the problems we have identified. This will require a concrete cost model for target NoSQL databases, a method of generating candidate views, and a method for searching through the space of possible schema. We expect such a tool to arrive at more efficient solutions than a human operator, while requiring only minimal configuration. In addition to comparing output of our tool against schemata chosen by human operators, we also intend to compare against a baseline schema analogous to a normalized schema which would be used in a relational databases.

In addition, our representation is not currently rich enough to take advantage of all the features of some data stores. For example, document stores such as MongoDB allow arbitrary nesting of hashes which could be useful for embedding one related entity within another. We expect to be able to enrich our model to take advantage of these capabilities.

6. CONCLUSION

Proper schema design is critical for high performance of NoSQL databases. Tools exist to assist with schema design for relational databases, but NoSQL databases present several unique challenges. As with relational databases, denormalization is required for high performance in NoSQL databases, but comes at the penalty of increased storage. Also, unlike materialized views in relational databases, the application must maintain denormalized data manually.

Human operators cannot fully explore the space of schema choices due to the large space of options available and several competing tradeoffs. Examining possible schemata for a realistic workload suggests that this is true in practical scenarios. An automated tool which optimizes a schema for a given workload based on query and storage costs is a promising alternative.

7. ACKNOWLEDGEMENTS

We thank Rui Liu for some initial work exploring the space of NoSQL schema design. We also thank Hemant Saxena for early input into a cost model for Cassandra.

8. REFERENCES

[1] HBase: A Distributed Database for Large Datasets. Retrieved March 7, 2013 from http://hbase.apache.org.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB '00*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[3] Benoit Dageville, D. Das, K. Dias, K. Yagoub, and M. Zait. Automatic SQL tuning in oracle 10g. *VLDB '04*, 30:1098–1109, 2004.

[4] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. Static and dynamic semantics of NoSQL languages. In *POPL '13*, pages 101–114, New York, New York, USA, 2013. ACM Press.

[5] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[6] A. Calil and S. Mello. SimpleSQL : A Relational Layer for SimpleDB. In *Advances in Databases and Information Systems*, pages 99–110. 2012.

[7] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, May 2011.

[8] E. Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Sebastopol, CA, 2 edition, 2011.

[9] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35, Apr. 2010.

[10] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database : C-Store 7 Years Later. In *VLDB '12*, volume 5, pages 1790–1801, 2012.

[11] A. Rasin and S. Zdonik. An Automatic Physical Design Tool for Clustered Column-Stores. In *EDBT '13*, pages 203–214, 2013.

[12] G. L. Sanders and S. Shin. Denormalization effects on performance of RDBMS. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE Comput. Soc, 2001.

[13] 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. *Proceedings of the 2013 International Workshop on Testing the Cloud - TTC 2013*, 2013.

[14] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O. Neil, P. O. Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store : A Column-oriented DBMS. In *VLDB '05*, pages 553–564, 2005.

[15] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: a versatile tool for physical data independence. *The VLDB Journal The International Journal on Very Large Data Bases*, 5(2):101–118, Apr. 1996.

[16] T. Vajk, L. Deák, K. Fekete, and G. Mezei. Automatic NoSQL Schema Development: A Case Study. In *Artificial Intelligence and Applications*, number Pdcn, pages 656–663. Actapress, 2013.

[17] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB '04*, pages 1087–1097, 2004.