NoSE: Schema Design for NoSQL Applications

Michael J. Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu

Abstract—Database design is critical for high performance in relational databases and a myriad of tools exist to aid application designers in selecting an appropriate schema. While the problem of schema optimization is also highly relevant for NoSQL databases, existing tools for relational databases are inadequate in that setting. Application designers wishing to use a NoSQL database instead rely on rules of thumb to select an appropriate schema. We present a system for recommending database schemas for NoSQL applications. Our cost-based approach uses a novel binary integer programming formulation to guide the mapping from the application's conceptual data model to a database schema. We implemented a prototype of this approach for the Cassandra extensible record store. Our prototype, the NoSQL Schema Evaluator (NoSE) is able to capture rules of thumb used by expert designers without explicitly encoding the rules. Automating the design process allows NoSE to produce efficient schemas and to examine more alternatives than would be possible with a manual rule-based approach.

Index Terms—NoSQL, physical design, schema optimization, workload modeling.

1 Introduction

NoSQL systems have become a popular choice as database backends for applications because of the high performance, scalability, and availability that they provide. In this paper, we focus on systems that Cattell termed *extensible record stores* in his taxonomy of NoSQL systems [1]. In these extensible record stores, applications can create tables of records, with each record identified by a key. However, the application need not define the set of columns in the records in advance. Instead, each record can have an arbitrary collection of columns, each with an associated value. Because of this flexibility, applications can encode their data in both the keys and column values. We refer to tables in such systems as *column families*. Examples of extensible record stores that support this column family model include Cassandra [2], HBase [3], and Bigtable [4].

Before a developer can build an extensible record store application, it is necessary to define a schema for the underlying record store. Although the schema of an extensible record store is flexible in the sense that the application does not need to define specific columns in advance, it is still necessary to decide what column families will exist in the record store, and what information each column family encodes. These choices are important because the performance of the application depends strongly on the underlying schema. For example, some schemas may provide answers to queries with a single lookup while others may require multiple requests to the extensible record store.

Although it is important to choose a good schema, there are no tools or established methodologies to guide and support this process. Instead, schema design for extensible record stores is commonly based on general heuristics and rules of thumb. For example, eBay [5] and Netflix [6] have shared examples and general guidelines for designing schemas for Cassandra. Specific recommendations include *not* designing column families as one would design relational tables, ensuring that column families reflect the anticipated workload, and denormalizing data in the record store. While such recommendations are useful, they are necessarily vague and generic, and require adaptation to each application.

In this paper, we propose a more principled approach to the problem of schema design for extensible record stores. Our objective is to replace general schema design rules of thumb with a tool that can recommend a specific schema optimized for a target application. Our tool uses a cost-based approach. By estimating the performance that candidate schemas would have for the target application, we recommend the schema that results in the best estimated performance. We designed our tool for use early in the application development process; the tool recommends a schema and the application is then developed using that schema. In addition to providing a schema definition, our tool also recommends a specific implementation of the application's queries against the proposed schema.

This work makes the following contributions. First, we formulate the *schema design problem* for extensible record stores. Second, we propose a solution to the schema design problem embodied in a schema design advisor we call *NoSE*, the *NoSQL Schema Evaluator*. We start with a conceptual model of the data required by a target application as well as a description of how the application will use the data. NoSE then recommends both an extensible record store schema, i.e., a set of column family definitions optimized for the target application, and guidelines for developing applications using this schema. Finally, we present a case study and evaluation of NoSE, using two application scenarios.

NoSE was originally presented in an earlier paper [7]. In this extended presentation, we described how NoSE supports a broader class of queries, namely those with acyclic query graphs. We also include additional information about costing and update plan generation, as well as a new application case study (Section 8).

- *M.J. Mior and K. Salem are with the University of Waterloo.*
- *A. Aboulnaga is with the Qatar Computing Research Institute.*
- *R. Liu is with HP Vertica.*

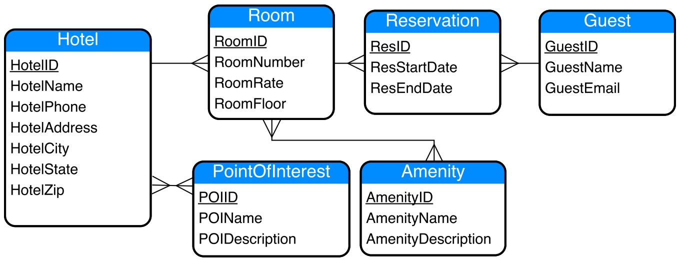


Fig. 1. Entity graph for a hotel booking system. Each box represents an entity set, and edges between boxes represent relationships.

2 Schema Design Example

In this section we present a simple example to illustrate the schema design problem for extensible record stores. Suppose we are building an application to manage hotel reservations. The conceptual model in Figure 1, adapted from Hewitt [8], describes the application's data.

The schema design problem for extensible record stores is the problem of deciding what column families to create and what information to store in each column family, for a given application. In general, this will depend on what the target application needs to do. For example, suppose that the application will need to use the extensible record store to obtain information about the points of interest (POIs) near hotels booked by a guest, given the guest's `GuestID`. The primary operations supported by an extensible record store are retrieval (`get`) or update (`put`) of one or more columns from a record, given a record key. Thus, an application could easily answer this query if the record store included a column family with `GuestIDs` as record keys and columns corresponding to POIs. That is, the column family would include one record for each guest. A guest's record includes one column for each POI associated with a hotel at which that guest has booked a room. The column names are `POIIDs`, and each column stores a composite value consisting of `POIName` and `POIDescription`. In general, each guest's record in this column family may have different columns. Furthermore, the application may add or remove columns from a guest's record when updating that guest's hotel bookings in the record store. With such a column family, the application can obtain point of interest information for a given guest using a single `get` operation. This column family is effectively a materialized view which stores the result of the application query for all guests.

In this paper, we will describe such a column family using the following triple notation:

```
[GuestID][POIID][POIName, POIDescription]
```

The first element of the triple indicates the attribute values used as record keys in the column family. The second element indicates the attribute values used as column names, and the third indicates those used as column values. We refer to the first element as the *partitioning key*, since extensible record stores typically horizontally partition column families based on the record keys. We refer to the second element as the *clustering key*, since extensible record stores typically physically cluster each record's columns by column name. We assume records in each partition are sorted according to the clustering key.

Although this column family is ideal for executing the single application query we have considered, it may not be ideal when we consider the application's entire workload. For example, if the application expects to be updating the names and descriptions of points of interest frequently, the above column family may be not be ideal because of the denormalization of point of interest information, i.e., the name and description of a POI may appear multiple times in the records for different guests. Instead, it may be better to create two column families, as follows:

```
[GuestID][POIID][]
[POIID][][POIName, POIDescription]
```

This stores information about each point of interest once, in a separate column family, making it easy to update. Similarly, if the application also needs to perform another query that returns information about the points of interest near a given hotel, it may be better to create three column families, such as these:

```
[GuestID][HotelID][]
[HotelID][POIID][]
[POIID][],[POIName, POIDescription]
```

In this schema, which is more normalized, records in the third column family consist of a key (a `POIID`) and a single column which stores the `POIName` and `POIDescription` as a composite value. The second column family, which maps `HotelIDs` to `POIIDs`, will be useful for both the original query and the new one.

The goal of our system, NoSE, is to explore this space of alternatives and recommend a good set of column families, taking into account both the entire application workload and the characteristics of the extensible record store.

NoSE solves a schema design problem similar to the problem of schema design for relational databases. However, there are also significant differences between the two problems. Relational systems provide a clean separation between logical and physical schemas. Standard procedures exist for translating a conceptual model, as in Figure 1, to a normalized logical relational schema, i.e., a set of table definitions, usable for defining the application's workload. The logical schema often determines a physical schema consisting of a set of base tables. The physical layout of these base tables is then optimized and supplemented with additional physical structures, such as indexes and materialized views, to tune the physical design to the anticipated workload. There are many tools for recommending a good set of physical structures for a given workload [9], [10], [11], [12], [13], [14], [15], [16].

Extensible record stores, in contrast, do not provide a clean separation between logical and physical design. There is only a single schema, which is both logical and physical. Thus, NoSE starts with the conceptual model, and produces both a recommended schema and plans for implementing the application against the schema. Further, the schema recommended by NoSE represents the entire schema, not a supplement to a fixed set of base tables. Unlike most relational physical design tools, NoSE must ensure that the workload is *covered*, i.e., that the column families it recommends are sufficient to allow for the implementation

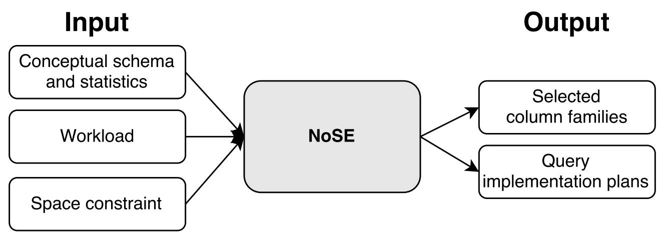


Fig. 2. Schema advisor overview

```
SELECT Guest.GuestName, Guest.GuestEmail FROM
  Guest.Reservation.Room.Hotel WHERE
  Hotel.HotelCity = ?city AND
  Room.Amenity.AmenityName = ?amenity AND
  Room.RoomRate > ?rate
```

Fig. 3. An example query against the hotel booking system schema

of the entire workload. We provide further discussion of relational physical design tools in Section 10.

3 System Overview

Figure 2 gives a high level illustration of the NoSE schema advisor. We designed NoSE for use early in the process of developing an extensive record store application. The advisor produces two outputs. The first is a recommended *schema*, which describes the column families used to store the application's data. The second output is a set of *plans*, one plan for each query and update in the workload. Each plan describes how the application should use the column families in the recommended schema to implement a query or an update. These plans serve as a guide for the application developer.

3.1 Database Conceptual Model

To recommend a schema for the target application, NoSE must have a conceptual model describing the information to store in the record store. NoSE expects this conceptual model in the form of an *entity graph*, such as the one shown in Figure 1. Entity graphs are simply a restricted type of entity-relationship (ER) model [17]. Each box represents a type of entity, and each edge is a relationship between entities and the associated cardinality of the relationship (one-to-many, one-to-one, or many-to-many). Entities have attributes, with one of these serving as a key. For example, the model shown in Figure 1 indicates that each room has a room number and rate. In addition, each room has is associated with a hotel and set of reservations.

3.2 Workload Description

The target application's workload is a set of parameterized query and update statements. Each query and update has an associated weight indicating its relative frequency in the anticipated workload. We focus here on the queries, and defer discussion of updates to Section 7.

Each query in the workload returns information about one or more entities in the entity graph. Figure 3 shows an example of a NoSE query, expressed using an SQL-like syntax, which returns the names and email addresses of guests who have reserved rooms with a particular amenity in given city at a given minimum rate. In this example, `?city`, `?amenity` and `?rate` are parameters. NoSE expresses queries directly over the conceptual model. Specifically, each query implies a *query graph* which is a subgraph of the entity graph. A path referenced in the `FROM` clause defines a portion of this graph. A query can define branches in the graph by specifying additional paths for attributes in the `SELECT` or `WHERE` clauses. The current implementation of NoSE is restricted to acyclic query graphs. However, this is not a fundamental limitation to our approach. We note that queries are required to have an equality predicate on the first entity set in the `FROM` clause in order to construct a valid `get` request to the underlying datastore. Figure 4 shows an example of a query graph.

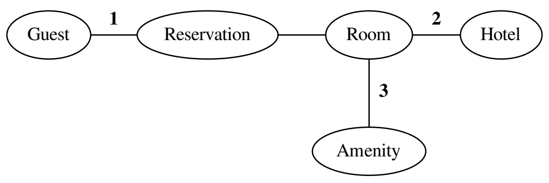


Fig. 4. Query graph for the query in Figure 3. We describe the edge labels in Section 4.1.2.

To define the semantics of these queries, we must consider which tuples a query produces. Conceptually, we consider the tuples produced by the join of all the relations in the query graph using the associated relationships between entity sets. These tuples are then filtered using the predicates given in the query. This means that a query returns data about a particular entity if there exists a series of related entities in the query graph that together satisfy the predicate of the query. Query results retain any duplicates in the resulting list of tuples.

We emphasize that the underlying extensible record store supports only simple `put` and `get` operations on column families, and is unable to directly interpret or execute queries like the one shown in Figure 3. Instead, the application itself must implement queries such as this, typically using a series of `get` operations, perhaps combined with application-implemented filtering, sorting, or joining of results. Nonetheless, by describing the workload to NoSE in this way, the application developer can convey the purpose of a sequence of low-level operations, allowing NoSE to optimize over the scope of entire high-level queries, rather than being restricted to individual low-level optimizations. Of course, another problem with describing the application workload to NoSE in terms of `get` and `put` operations on column families is that the column families are not known. Indeed, the purpose of NoSE is to recommend a suitable set of column families for the target application.

Although it is not shown in Figure 3, NoSE queries can also specify a desired ordering on the query results, using an `ORDER BY` clause. This allows NoSE to recommend column families which exploit the implicit ordering of clustering keys to produce results in the desired order.

3.3 Extensible Record Stores

The target of our system is extensible record stores, such as Cassandra or HBase. These systems store collections of keyed records in column families. Records in a collection need not all have the same columns.

Given a domain $\mathcal{K}$ of partition keys, an ordered domain $\mathcal{C}$ of clustering keys, and a domain $\mathcal{V}$ of column values, we model a column family as a table of the form

$$\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V})$$

That is, a column family maps a partition key to a set of clustering keys, each of which maps to a value. Clustering keys provide an order for records in a single partition. For example, in Section 2, we used an example of a column family with `GuestID`s as partition keys, `POIID`s as clustering keys, and POI names and descriptions as values. Such a column family would have one record for each `GuestID`, with POI information for that guest's records clustered using the `POIID`.

We assume that the extensible record store supports only `put`, `get`, and `delete` operations on column families. To perform a `get` operation, the application must supply a partition key and a range of clustering key values. The `get` operation returns all $\mathcal{C} \mapsto \mathcal{V}$ pairs within the specified clustering key range, for the record identified by the partition key. For example, the application could use a `get` operation to retrieve information about the points of interest associated with a given `GuestID`. Similarly, a `put` operation can modify the $\mathcal{C} \mapsto \mathcal{V}$ pairs associated with a single partition key, and a `delete` operation deletes $\mathcal{C} \mapsto \mathcal{V}$ pairs associated with a given partition key.

Some extensible record stores provide additional capabilities beyond the three basic operations we have described. For example, in HBase it is possible to get information for a range of partition keys, since records are also sorted based on their partition key. As another example, Cassandra provides a limited form of secondary indexing, allowing applications to select records by something other than the partitioning key. However, many Cassandra applications do not use secondary indexes for performance reasons [18]. For simplicity, we restrict ourselves to the simple `get/put` model we have described, as it captures common functionality.

3.4 The Schema Design Problem

A schema for an extensible record store consists of a set of column family definitions. Each column family has a name as an identifier and its definition includes the domains of partition keys, clustering keys, and column values used in that column family.

Given a conceptual model (optionally with statistics describing data distribution), an application workload, and an optional space constraint, the schema design problem is to recommend a schema such that (a) each query in the workload is answerable using one or more `get` requests to column families in the schema, (b) the weighted total cost of answering the queries is minimized, and optionally (c) the aggregate size of the recommended column families is within a given space constraint. Solving this optimization problem is the objective of our schema advisor. In addition to the schema, for each query in the workload, NoSE recommends a specific *plan* for obtaining an answer to that query using the recommended schema. We discuss these plans further in Section 4.2.

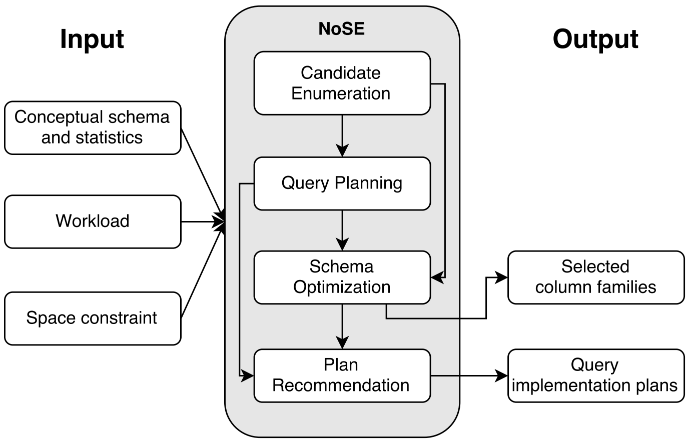


Fig. 5. Complete schema advisor architecture

4 Schema Advisor

Given an application's conceptual model and workload, as shown in Figure 2, NoSE proceeds through four steps:

1) **Candidate Enumeration** Generate a set of *candidate* column families, based on the workload. By inspecting the workload, the advisor generates only candidates which may be useful for answering the queries in the workload.
2) **Query Planning** Generate a space of possible implementation plans for each query. These plans make use of the candidate column families produced in the first step.
3) **Schema Optimization** Generate a binary integer program (BIP) from the candidates and plan spaces. The BIP is then given to an off-the-shelf solver (we have chosen to use Gurobi [19]) which chooses a set of column families that minimizes the cost of answering the queries.
4) **Plan Recommendation** Choose a single plan from the plan space of each query to be the recommended implementation plan for that query based on the column families selected by the optimizer.

Figure 5 illustrates this process. In the reminder of this section, we discuss candidate enumeration and query planning. Section 5 presents schema optimization and plan recommendation.

4.1 Candidate Enumeration

One possible approach to candidate enumeration is to consider all possible column families for a given set of entities. However, the number of possible column families is exponential in the number of attributes, entities, and relationships in the conceptual model. Thus, this approach does not scale well.

```
function: Materialize
input   : A query $q$
output  : A materialized view for the query
  // first entity equality predicates
1 $\mathcal{K} \leftarrow [c.attr \mid c \in q.where \wedge c.op = \text{'='}$
      $\wedge\, c.attr.entity = q.from[0]]$;
  // all other equality predicates
2 $\mathcal{C} \leftarrow [c.attr \mid c \in q.where \wedge c.op = \text{'='}$
      $\wedge\, c.entity \neq q.from[0]]$;
  // add all other predicates
3 $\mathcal{C} \leftarrow \mathcal{C} + [c.attr \mid c \in q.where \wedge c.attr \notin \mathcal{K} \bigcup \mathcal{C}]$;
  // add ordering attributes
4 $\mathcal{C} \leftarrow \mathcal{C} + (q.order\_by \setminus \mathcal{C})$;
  // end with IDs from all entities
5 $\mathcal{C} \leftarrow \mathcal{C} + [e.id \mid e \in entities(q)$
      $\wedge\, e \notin \{a.entity \mid a \in \mathcal{C}\}]$;
  // selected attributes as values
6 $\mathcal{V} \leftarrow q.select \setminus \mathcal{K} \setminus \mathcal{C}$;
7 return $\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V})$;
```

Alg. 1. Materialized view column family generation

Instead, we enumerate candidates using a two-step process based on the application's workload. First, we independently enumerate a set of candidate column families for each query in the application workload. The union of these sets is the initial candidate pool. Second, we supplement this pool with additional column families constructed by combining candidates from the initial pool. The goal of the second step is to add candidates which are likely to be useful for answering more than one query while consuming less space than two separate column families. We do not claim that NoSE's candidate enumerator guarantees the enumeration of column families which result in an optimal schema However, the optimization process we discuss in Section 5 chooses an optimal subset of the enumerated candidates for the given cost model. NoSE's candidate enumerator is pluggable and could be replaced with any enumerator which produces valid column families capable of answering queries in the given workload. We leave other heuristics to determine additional useful column families as future work.

4.1.1 Candidate Column Families

Recall from Section 3.3 that a column family is a mapping of the form $\mathcal{K} \mapsto (\mathcal{C} \mapsto \mathcal{V})$. To define a specific column family, we need to determine $\mathcal{K}$, $\mathcal{C}$, and $\mathcal{V}$. That is, we need to specify what the keys, columns, and values will be for the column family.

We consider column families in which keys, columns, and values consist of one or more attributes from the application's conceptual model. We represent each column family as a triple, consisting of a set of partition key attributes, an ordered list of clustering key attributes, and a set of value attributes. For example, we can define a column family useful for retrieving, for a given city and state, a list of hotel names, addresses, and phone numbers, in order of hotel name. We represent this column family by the following triple:

```
[HotelCity, HotelState][HotelName,
   HotelID][HotelAddress, HotelPhone].
```

Column families are not limited to containing information on a single entity from the conceptual model. For any query in our language, we can define a column family useful for directly retrieving answers to that query, which we call a *materialized view*. Algorithm 1 describes how NoSE generates materialized views from queries. For example, the query shown in Figure 3, which returns the names and emails of guests who have reserved rooms at hotels in a given city, at room rates above a given rate with a given amenity, corresponds to the following materialized view:

```
[HotelCity, AmenityName][RoomRate, AmenityID,
  HotelID, RoomID, ResID, GuestID][GuestName,
  GuestEmail]
```

By supplying city and amenity names, and a minimum room rate, an application can use this column family to retrieve a list of tuples, each of the form `(RoomRate, AmenityID, HotelID, RoomID, ResID, GuestID,GuestName,GuestEmail)`. Each tuple corresponds to a distinct room reservation of a hotel room with the specified amenity and minimum room rate. The query returns tuples in order of `RoomRate`.

4.1.2 Per-Query Candidate Enumeration

The schema optimizer requires flexibility in the choice of column families since its space budget may not allow the recommendation of a materialized view for each query. In addition, when we later consider updates, a column family for each query may become too expensive to maintain. Therefore, in addition to the materialized view, the enumerator also includes additional column families provide partial answers for each query. The application can use these to answer the query by combining the results of multiple `get` requests to different column families.

To generate the full pool of candidate column families for a given query, we decompose the query at each possible edge in the query graph. Decomposing a query at a specific edge in the query graph splits the query into two parts, which we call the *prefix query* and the *remainder query*. Later, when constructing a query plan, the planner joins these decomposed query graphs along the cut edges to produce a complete plan for a query. Figure 6 illustrates the first level of this recursive splitting process for the example query from Figure 3. We show the decomposition for just the three labelled edges in (Figure 4).

For each of the generated prefix/remainder queries, NoSE first enumerates its materialized view. If the `SELECT` clause of the prefix query includes non-key attributes, NoSE enumerates two additional candidates: one that returns only the key attributes, and a second that returns required non-key attributes given the key. For example, for the query in Figure 3, in addition to the materialized view, the enumerator will also generate the following two candidates:

```
[HotelCity,AmenityName][RoomRate,AmenityID,
 HotelID, RoomID, ResID, GuestID][]
[GuestID][][GuestName,GuestEmail]
```

Decomposition Edge	Prefix query	Remainder query
1	`SELECT Reservation.ResID FROM Reservation.Room.Hotel WHERE Hotel.HotelCity = ? AND Room.Amenity.AmenityName = ? AND Reservation.Room.RoomRate > ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.ResID = ?`
2	`SELECT Hotel.HotelID FROM Hotel WHERE Hotel.City = ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest.Reservation.Room.Hotel WHERE Hotel.HotelID = ? AND Room.RoomRate > ? AND Room.Amenity.AmenityName = ?`
3	`SELECT Amenity.AmenityID FROM Amenity WHERE Amenity.AmenityName = ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest.Reservation.Room.Hotel WHERE Hotel.HotelCity = ? AND Room.Amenity.AmenityID = ? AND Room.RoomRate > ?`

Fig. 6. Example of query decomposition for candidate enumeration for the query in Figure 3

The former is useful for returning a set of `GuestID`s, given a city, an amenity, and a room rate, and the latter can then produce the guests' names and email addresses.

Finally, the enumerator may generate additional candidates corresponding to *relaxed* versions of the prefix query. Specifically, when the enumerator considers a query of the form

```
SELECT attributes FROM path-prefix WHERE
path.attr op ? AND predicate2 AND ...
```

it also generates materialized views for relaxed queries of the form

```
SELECT attributes, attr FROM
path-prefix WHERE predicate2 AND ...
```

That is, the enumerator removes one or more predicates and adds the attributes involved in the predicates to the `SELECT` clause.

Predicates are only considered for removal if the remaining query will have at least one equality predicate remaining. (The application will require this to construct a valid `get` request on the column family in the recommended plan.) NoSE also relaxes queries involving ordering in the same way, by moving an attribute in an `ORDER BY` clause to the `SELECT` list.

The full enumeration algorithm is given in Algorithm 2. For a query with k edges, this algorithm will generate at least $N_k = 1 + k + \sum_{j=1}^{k-1} N_j = 2^{k+1} - 1$ candidate column families, ignoring any relaxed prefix queries. The number of relaxed prefix queries is exponential in the number of attributes occuring in the `WHERE` and `ORDER BY` clauses, since `Relax(`p`)` considers all subsets of those attributes. Thus, the number of candidates per query grows exponentially with both the size of the query graph and the size of the `WHERE` and `ORDER BY` clauses.

4.1.3 Candidate Combinations

Once the enumerator has produced candidates for each query in the workload, it then generates additional candidates by combining the per-query candidates that are already present in the pool. Specifically, the enumerator looks for pairs of column families for which both have the same partition key ($\mathcal{K}$), neither has a clustering key ($\mathcal{C}$), and each has different data attributes ($\mathcal{V}$). For each such pair, the enumerator generates an additional candidate that has the same partition keys and all the data attributes from both of the column families it identified. Thus, the new candidate will be larger than either of the original candidates but will be potentially useful for answering more than one query.

```
function: Enumerate
input   : A query $q$
output  : Enumerated column families for $q$
// create a materialized view
1 $C \leftarrow \{$Materialize$(q)\}$;
2 foreach edge $e$ in $q.graph$ do
3     // split prefix and remainder
4     $p, r \leftarrow$ Decompose$(q, e)$;
      // add relaxed prefix queries
5     $C \leftarrow C \bigcup \{$Materialize$(q') \mid q' \in$ Relax$(p)\}$;
      // materialize prefix and recurse
6     $C \leftarrow C \bigcup \{$Materialize$(p)\} \bigcup$ Enumerate$(r)$;
7 return $C$;
```

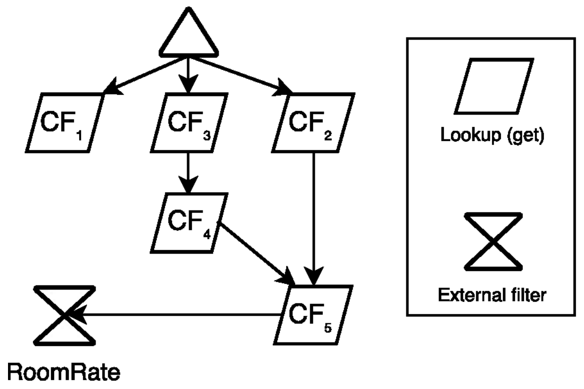
`Decompose` splits the query graph in two on the given edge.
`Relax` produces relaxed prefix queries as described in the text.

Alg. 2. Column family enumeration

There are additional opportunities for creating new column families by combining candidates from the pool, but NoSE's enumerator currently only exploits this one type of combination. Increasing the number of candidates increases the opportunity for the schema advisor to identify a high-quality schema (i.e. one with lower cost) but this also increases the running time of the advisor. As future work, we intend to explore other opportunities for candidate generation in light of this tradeoff as well as heuristics to prune column families which are unlikely to be useful.

4.2 Query Planning

One component of the output of NoSE is a *query execution plan* for each query in the input workload. Query execution plans consist of a series of three possible steps: (a) a `get` or `put` request to the underlying data store, (b) filtering of data fetched from the data store by the application, or (c) sorting of data by the application. Each of these operations has an associated cost, which we describe in Section 6. These plans



CF_1	[HotelCity][AmenityID, RoomRate, RoomID][]
CF_2	[HotelCity][AmenityID, RoomID][]
CF_3	[HotelCity][HotelID][]
CF_4	[HotelID][AmenityID, RoomID][]
CF_5	[RoomID][][RoomRate]

Fig. 7. Example query plan space

describe to application developers how each query should be implemented.

The task of the query planner is to enumerate all possible plans for evaluating a given query, under the assumption that all candidate column families are available. Each plan is a sequence of steps, using candidate column families, that will produce an answer to an application query. We refer to the result of this process as the *plan space* for the given query. Later, during schema optimization, the schema advisor will use the plan spaces for each query to determine which candidate column families to recommend.

NoSE performs query planning as part of the same recursive decomposition process that generates candidate column families. Consider the decomposition of the running example query (Figure 3) shown in Figure 6. For each prefix query, the query planner generates a set of implementation plans, each of which starts by retrieving the results of the prefix query, and finishes by joining those results to a (recursively calculated) plan for the corresponding remainder query. When generating plans for a prefix query, the planner will generate one set of plans for each candidate column family generated for that prefix query, and for any other candidate column families that subsume a candidate for the prefix query. In general, because query planning is based on the same recursive decomposition used to enumerate candidate column families (Algorithm 2), the size of the plan space for a query with k edges in its query graph grows exponentially with k.

Figure 7 shows the plan space for the query below:

```
SELECT Room.RoomID FROM Room WHERE
Room.Hotel.HotelCity = ?city AND
Room.Amenity.AmenityID = ?amenityID AND
Room.RoomRate > ?rate
```

There are three possible plans in the plan space. The first uses the materialized view CF_1 to answer the query directly. The second finds the HotelID for all hotels in a given HotelCity using CF_3. The HotelID is then used to find all the RoomIDs for the given hotel using CF_4. Finally, application discovers the RoomRates using CF_5 and filters the RoomIDs to only contain those matching the predicated on RoomRate. The final plan is similar, but goes directly from a HotelCity to a list of RoomIDs using CF_2.

When developing a query plan, it is necessary to select an order in which to execute each individual query and join the values based on the IDs of each entity. Since we require at least one equality predicate for each get request, we start with the equality predicate in the query graph with the lowest cardinality. Entities in the query graph are then selected based on those reachable from the currently joined entities and in order by increasing cardinality. While this is not guaranteed to be optimal, choosing entities in this order reduces the size of intermediate result sets. There are many possible alternative heuristics [20] but we use this for simplicity.

minimize $\sum_i \sum_j f_i C_{ij} \delta_{ij}$

subject to

All used column families being present

$\delta_{ij} \le \delta_j, \forall i, j$

Maximum space usage S

$\sum_j s_j \delta_j \le S$

Plus per-query plan graph constraints (see text)

Fig. 8. Binary integer program for schema optimization

5 Schema Optimization

A naïve approach to schema optimization is to examine each element in the power set of candidate column families and evaluate the cost of executing each workload query using a plan that involves only the selected candidates. However, this approach scales poorly as it is exponential in the total number of candidate column families.

Papadomanolakis and Ailamaki [14] present a more efficient approach to the related problem of index selection in relational database systems. Their approach formulates the index selection problem as a binary integer program (BIP) which selects an optimal set of indices based on the index configurations that are useful for each query in the workload. Their approach uses a set of decision variables for each query, with the number of variables per query equal to the number of combinations of indices useful to that query. This is still exponential, like the naïve approach, but only in the number of indices relevant to each query, rather than the total number of candidate indices.

Like Papadomanolakis and Ailamaki, we have implemented schema optimization by formulating the problem as a BIP. However, because of the simple structure of the query implementation plans that our schema advisor considers, we are able to provide a simpler formulation for our problem.

Our schema advisor uses the query plan spaces described in Section 4.2 to generate a BIP. A binary decision

$$\begin{aligned}
\delta_{1,4} &\leq \delta_{1,3} \\
\delta_{1,5} &\leq \delta_{1,2} + \delta_{1,4} \\
\delta_{1,j} &\leq \delta_j \forall j \in \{1,2,3,4,5\} \\
\delta_{1,1} + \delta_{1,2} + \delta_{1,3} &= 1 \\
\delta_{1,1} + \delta_{1,2} + \delta_{1,4} &= 1 \\
\delta_{1,1} + \delta_{1,5} &= 1
\end{aligned}$$

Fig. 9. BIP constraints for the plan graph from Figure 7

variable, δ_{ij}, exists for each combination of a candidate column family and a workload query. The variable δ_{ij} indicates whether the ith query will use the jth column family in its implementation plan. The objective of the optimization program is to minimize the quantity $\sum_i \sum_j f_i C_{ij} \delta_{ij}$, where C_{ij} represents the cost of using the jth column family in the plan for the ith query and f_i is the query frequency. However, after solving this optimization problem, we run the solver again with an additional constraint that the cost of the workload equals the minimum value which was just discovered, and with the objective of minimizing the total number of column families in the recommended schema. This allows NoSE to produce the schema with the smallest number of column families out of the set of those which are most efficient.

In addition to the decision variables δ_{ij}, our program formulation uses one other decision variable per candidate column family. These variables indicate whether the solution includes the corresponding column families in the set recommended by the schema advisor. We use δ_j to represent this per-column-family decision variable for the jth candidate column family. Our BIP includes constraints that ensure that

- the solution includes the jth column family in the recommendation if the solution uses it in the plan for at least one query, and
- (optionally) that the total size of the recommended column families is less than the specified space constraint.

To allow sorting to occur at any point in query execution, we also add a constraint that results are properly sorted. Overall, this approach requires $|Q||P|$ variables representing the use of column families in query implementation plans, and $|P|$ variables representing candidate column families, where $|Q|$ represents the number of queries and $|P|$ is the number of candidate column families (Section 4.1.2). We also allow an optional storage constraint whereby the user can specify a limit S on the amount of storage occupied by all column families. The estimated size of each column family s_j is also given as a parameter to the BIP. Figure 8 summarizes the binary integer program.

As noted in Figure 8, the BIP also requires a set of *plan graph constraints*, on the variables δ_{ij}, which ensure that the solver will choose a set of column families for each query that correspond to one of the plans in the query plan space. These constraints derive from the per-query plan spaces determined by the query planner. For example, in Figure 7, the solution can select at most one of CF_1, CF_3, and CF_2 to answer this query, since each is useful for different plans, and the solution selects only one plan per query. In addition, if the solution selects CF_3, then it must also select CF_4 and CF_5. The BIP will include corresponding constraints on the decision variables δ_{ij} that indicate whether the solution will use those column families to answer this query. Figure 9 shows the plan graph constraints for the example shown in Figure 7.

After solving the BIP, making the final plan recommendation is straightforward. There is a unique plan with minimal cost based on the values of the decision variables in the BIP.

6 Cost Model

The BIP constants C_{ij} represent the cost of using a particular column family in the plan for a particular query. For the example shown in Figure 7, there will be five such constants, one for each column family node in the plan graph. As it generates the BIP, NoSE uses its cost model to determine values for these constants.

Each lookup node in the plan graph represents one or more `get` operations against a particular column family. The corresponding BIP constant represents the total cost of all such get operations. NoSE estimates each node's total cost using a two-parameter cost function $T(n, w)$, where n represents the number of `get` operations that the plan will perform against the column family, and w represents the "width" of each request, i.e., the number of $\mathcal{C} \mapsto \mathcal{V}$ pairs that will be returned by each `get`.

NoSE's plans involve only a single `get` from the first column family in each plan. Thus, $n = 1$ for the first column family. The number of `get` operations for the next column family depends on the number of results returned from the `get` on the first column family. Thus, to estimate values of n for each column family in a plan, NoSE first estimates the result cardinality of the preceding column family - much as in join size estimation in relational database systems. For these cardinality estimates, NoSE currently makes use of simple statistical metadata that is described in terms of the conceptual model and provided as part of NoSE's input. Specifically, a user can provide the cardinality of each attribute in each entity set to NoSE. In addition, NoSE makes use of relationship cardinality constraints from the conceptual model and simple uniformity assumptions. NoSE also uses this same metadata, as well as the properties of the query, to estimate the value of w for each column family. This is a very simplistic approach to cardinality estimation. However, cardinality estimation (and costing in general) is not the focus this work, and NoSE's current approach could easily be replaced by a more sophisticated one.

NoSE's query plans may also include application-side filtering and sorting operations, in addition to column family access. Currently, NoSE's cost module treats filter operations as free. Since the query predicates are simple to evaluate and the application can perform filtering "on the fly" as the underlying record store returns results, filtering adds little to no overhead to the time required to retrieve the records. To account for the cost of application-side sorting, NoSE adds a small constant sorting penalty to the estimated

```
INSERT INTO Reservation SET ResEndDate = ?date

DELETE FROM Guest WHERE
  Guest.GuestID = ?guestID

UPDATE Reservation FROM Reservation.Guest
  SET Reservation.ResEndDate = ?
  WHERE Guest.GuestID = ?guestID

CONNECT User(?userID) TO Reservations(?resID)

DISCONNECT User(?userID)
  FROM Reservations(?resID)
```

Fig. 10. Example NoSE update statements

Query

```
SELECT Room.RoomRate FROM
Room.Hotel.PointsOfInterest
WHERE Room.RoomFloor = ?floor
AND PointsOfInterest.POIID = ?poiID
```

Materialized View Column Family

```
[Room.RoomFloor][PointsOfInterest.POIID,
    Hotel.HotelID, Room.RoomID]
    [Room.RoomRate]
```

Update

```
UPDATE Room FROM Room.Reservations.Guest
SET RoomRate = ?rate1 WHERE
Guest.GuestID = ?id AND
Room.RoomRate = ?rate2
```

Fig. 11. An example NoSE query, materialized view, and update

cost of the preceding column family in the plan. A more sophisticated model could adjust this penalty based on result size, and could more accurately account for the overlap of application-side sorting time with retrieval time.

6.1 Calibration

The actual cost of performing a plan's `get` operations on a column family depends on the performance characteristics of the underlying extensible record store. Therefore, NoSE learns the cost estimation function ($T(n, w)$) using an offline calibration process.

The calibration process uses a synthetic database consisting of set of column families with differing widths. We performed a series of experiments, by choosing a column family with a particular width w, performing n `get` requests against that column family, and measuring the total time required for the requests. Each request retrieves all w columns for a randomly chosen key. Thus, each such experiment provides a measured execution time for a particular n and w. We performed many of these experiments, and used linear regression over the results to determine $T(n, w)$.

7 Updates

The previous sections described how NoSE functions on a read-only workload, but it is important to also consider updates in the workload description. Updates implicitly constrain the amount of denormalization present in the generated schema. This effect results from the maintenance required when the same attribute appears in multiple column families. Each column family containing an attribute modified by an update is also modified, so repetition of attributes increases update cost.

We first introduce extensions to our workload description to express updates. We then describe the update execution plans that NoSE generates and recommends to the application developer. Finally, we describe modifications required to the enumeration algorithm and the BIP used by NoSE to support these updates.

7.1 Update Language

In order to support updates to the workload, we extend our query language with additional statements which describe updates to data in terms of the conceptual model, as illustrated in Figure 10. `INSERT` statements create new entities and result in insertions to column families containing attributes from that entity. We assume that the `INSERT` statement provides the primary key of each entity, but all other attributes are optional. `UPDATE` statements modify attributes, resulting in updates to any corresponding column families in the schema. `DELETE` statements remove all data about deleted entities from any associated column families. Both `UPDATE` and `DELETE` statements specify the entities to modify using the same predicates available for queries. Finally, `CONNECT` and `DISCONNECT` statements create or destroy relationships between entities. These statements simply specify the primary key of each entity and the relationship to modify. We also allow the creation of relationships on the insertion of a new entity by specifying foreign keys of related entities.

7.2 Update Plans

As with queries, NoSE must provide an implementation plan for each update, using the `get`, `put` and `delete` operations supported by the extensible record store. Because NoSE may denormalize attributes across multiple column families, it must first determine which column families are affected by the update, and then generate a plan for modifying each of those column families to reflect the changes.

In general, a NoSE update statement might not contain enough information to allow NoSE to construct an update plan. To illustrate this problem, suppose that the schema recommended by NoSE includes the materialized view shown in Figure 11. Suppose further that the workload includes the `UPDATE` shown in the figure. Such an `UPDATE` may affect the materialized view since the `UPDATE` changes the room rates stored in the view. Thus, the recommended plan for this update should include making changes to the materialized view, using `put` or `delete` operations. However, a plan cannot change room rates in this view without knowing the `RoomFloor`, `POIID`, `HotelID`, and `RoomID` associated with the room rates that need to change. This information is not provided by the `UPDATE`.

To resolve this problem, NoSE update plans may include *support queries*, which obtain information that the plan requires in order to perform the update. NoSE generates such support queries automatically, as needed, as part of its

```
function: Support
input   : A modification $u$ and a column family $t$
output  : A set of support queries

// get required attributes
1  $A \leftarrow \texttt{Required}(t, u)$;
2  if $|A| == 0$ then return $\emptyset$;

3  if $u.type \in \{\texttt{Insert}, \texttt{Connect}, \texttt{Disconnect}\}$ then
4      // split on relevant edges
5      $E \leftarrow c \mid c \in u.connections \land c \in t.graph$;
6      $G = \texttt{Split}(t.graph, E)$;

       // build a query for each subgraph
7      $S \leftarrow \emptyset$;
8      foreach subgraph $g$ in $G$ do
9          $A' \leftarrow \{a \mid a \in A \land a.entity \in g\}$;
10         $w' \leftarrow \{c \mid c.attr.entity \in g\}$;
11         if $|A'| = 0$ then continue;
12         $S \leftarrow S \bigcup \{(u.from, A', w', [])\}$;
13     return $S$;
14 else return $\{(u.from, A, u.where, [])\}$;
```

`Required` produces the necessary fields for the update.
`Split` divides a graph in two on a given edge.

Alg. 3. Support query generation

```
SELECT Room.RoomID
FROM Room.Reservations.Guest
WHERE Guest.GuestID = ?id
AND Room.RoomRate = ?rate2

SELECT Room.RoomFloor, Room.Hotel.HotelID,
PointsOfInterest.POIID
FROM PointsOfInterest.Hotels.Rooms
WHERE Room.RoomID = ?id
```

Fig. 12. Support queries for the update shown in Figure 11

planning and optimization process. Given an update from the workload and candidate column family, Algorithm 3 describes how NoSE generates the support queries necessary to update the column family. If an update includes all the information required to update a column family, then Algorithm 3 does not generate any support queries.

7.2.1 Plans for UPDATE

We will use the materialized view and UPDATE from Figure 11 to illustrate how NoSE generates plans for updates. For this UPDATE and view, NoSE will generate two support queries. The first will return RoomIDs for the rooms whose rates the UPDATE modifies. NoSE executes the second query once for each RoomID returned by the first query. The query returns the RoomFloor, HotelID, and POIIDs for the given RoomID. Figure 12 illustrates these two support queries. The results of this second query identify the record keys (RoomFloors) and columns the UPDATE needs to modify RoomRate in the materialized view. The plan recommended by NoSE can then perform each update using a put command against the materialized view.

In general, NoSE may require more than two support queries to update a column family, although two is sufficient in our example. The number of support queries that NoSE generates depends on the structure of the materialized view's query graph, and on which entity NoSE is updating. Furthermore, applying the updates to the column family once the necessary records and columns have been identified is not always as simple as our example suggests. If an UPDATE modifies an attribute used as part of the column names or part of the record key in the view, then NoSE cannot simply put the new value as it does in our previous example. Instead, such updates delete the old record or column and then insert a replacement.

7.2.2 Plans for DELETE

Plans for DELETE statements are similar to those for UPDATEs, since both types of statement affect a single type of entity in the conceptual model. Like an UPDATE, a DELETE may require support queries to determine which records to remove from an affected column family.

7.2.3 Plans for INSERT

If an INSERT statement does not include any CONNECT clauses, then NoSE only needs to update column families that contain only attributes of the newly inserted entity. In this case, the INSERT supplies all the necessary attribute values, and NoSE does not need to generate any support queries.

If an INSERT does include CONNECT clauses, then NoSE may need to update column families that include attributes from multiple types of entities, including the type of the inserted entity. Since the INSERT statement only specifies values for the attributes of the inserted entity, NoSE will construct one or more support queries to obtain the attribute values for other entities that appear in the column family. Furthermore, since the new entity's attribute values may be denormalized in the column family, the support queries determine how many new records or columns to add to the column family to reflect the addition of the new entity. For example, an INSERT of a new POI, with a CONNECT to a nearby hotel, may result in the addition of multiple columns in multiple records of the materialized view shown in Figure 11. Specifically, there will be a new column for each room in the hotel linked to the new POI. Support queries determine the RoomID, RoomFloor, and RoomRate of these rooms, so that the INSERT plan can add the necessary columns to the column family.

7.2.4 Plans for CONNECT and DISCONNECT

CONNECT and DISCONNECT may modify a column family if the column family's underlying query graph includes the edge that is being connected or disconnected. CONNECT statements may cause new records or columns to be inserted, and support queries obtain the necessary attribute values, much as was done for INSERT. Similarly, DISCONNECT may cause records or columns to be removed, and support queries determine the affected records and columns. When modeling the cost of CONNECT and DISCONNECT, we treat these as insertions or deletions to column families involving the relevant edge.

```
function: UpdateEnumerate
input   : A set of queries $Q$ and updates $U$
output  : A set of enumerated column families
  // enumeration for workload queries
1 $C \leftarrow \{$Enumerate($q$) $\mid q \in Q\}$;
  // enumeration for support queries
2 do twice
3     $C' \leftarrow C$;
4     foreach update $u$ in $U$ do
5         foreach column family $c$ in $C'$ do
6             if Modifies?($u$, $c$) then
7                 foreach query $q$ in Support($u$, $c$) do
8                     $C \leftarrow C \bigcup$ Enumerate($q$);
9 return $C \bigcup$ Combine($C$);
```

The Enumerate and Combine functions represent the candidate enumeration and candidate combination methods for query-only workloads from Section 4.1. Support is the support query generation algorithm (Algorithm 3). Modifies? tests whether an update requires modifications to a given column family.

Alg. 4. Column family enumeration for workloads with updates

$$\textbf{minimize} \sum_i \sum_j f_i C_{ij} \delta_{ij} + \sum_m \sum_n f_m C'_{mn} \delta_n$$

subject to

All constraints from Figure 8

Additional constraints per support query (see text)

Fig. 13. BIP modifications for updates

7.3 Column family enumeration for updates

Additional column families may be necessary to answer support queries for updates in the workload. When there are updates, the candidate enumerator uses the procedure shown in Algorithm . This procedure extends the candidate enumeration procedure for query-only workloads, which was originally described in Section 4.1. As shown in Algorithm , NoSE performs candidate enumeration for each query in the original workload, and twice for support queries. This is because support queries generated on the first iteration may cover new edges in the entity graph. Candidate column families for these support queries may themselves be affected by workload updates, resulting in support queries for support queries.

Update support queries increase the size of the application workload. For example, suppose that the original workload includes a query Q with a query graph of length k, and an update U that affects the first entity in Q's query graph. This will result in the addition of at least $k + 1$ support queries to the workload. This is because candidate enumeration for Q will generate $k + 1$ prefix queries involving the updated entity. Each of those queries will have a materialized view which is affected by U, and for which a support query will be required.

7.4 BIP Modifications

To incorporate updates into our BIP, we first add constraints for all support queries similarly to those for queries in the original workload. In addition, we add constraints to ensure that NoSE does not generate plans for support queries for a candidate column family unless that column family is part of the recommended design. The objective function receives an additional term, $\sum_m \sum_n f_m C'_{mn} \delta_n$, to represent the cost of updating each column family, which is contingent on the recommendation including this column family in the final schema. C'_{mn} is the cost of updating column family n for update m (with frequency f_m) given that the column family appears in the final schema (δ_n). The cost of support queries is also added using the same weight specified for the update in the workload. Figure 13 shows the modified BIP.

After solving this modified BIP, NoSE plans each update by first generating any necessary support query plans in the same way as plans for queries in the original workload. Each update plan then consists of a series of support query plans along with insertion or deletion as necessary for the update.

8 Case Study

In this section, we present an analysis of a partial workload extracted from EasyAntiCheat (EAC)[1], a real-time cheat detection engine for multiplayer games. Our goals are to illustrate the challenges of NoSQL schema design and to illustrate how NoSE works. In Section 9, we present a more quantitative evaluation of NoSE.

EAC receives large volumes of player behaviour data in real time. Their backend systems pull in this data and analyze player behaviour to determine patterns indicative of cheating. After hitting scalability limits with their relational database infrastructure, EAC considered Cassandra as a possible backend. Figure 14 shows a simplified version of the conceptual model for the application. Game servers have a number of player sessions with servers continually collecting information on the state of players in each session. The system stores data on millions of players and states, hundreds of thousands of player sessions and thousands of servers. Players generate new states at rates of up to several hundred thousand per second.

For this case study, we focus on a subset of the workload. Figure 15 shows the most important queries in the workload. The workload also includes updates (not shown in Figure 15) including the insertion of new player states, sessions, players, and servers. EAC estimates their workload to be roughly 80% writes and 20% reads. The majority of the writes come from the insertion of new states while most queries are instances of Q_1 and Q_2. We have assumed specific frequencies, fitting these constraints, for all queries and updates. For simplicity, we assume these are the only queries and updates performed by the system.

We used the EAC schema and workload as input to NoSE, and Figure 16 shows the five column recommended column families. The critical problem NoSE must resolve for this workload is how to store player states, which are voluminous and frequently inserted, and which are read by

1. www.easyanticheat.net

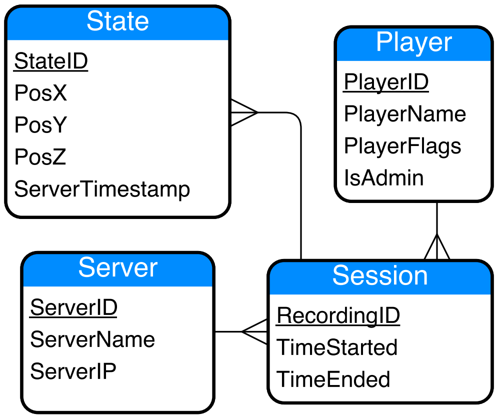


Fig. 14. Entities modeled in EasyAntiCheat

both Q_1 and Q_2. Furthermore, one of these queries retrieves states for a single player session, while the second retrieves states for all players on a game server. NoSE addresses this problem by recommending a single column family (CF_1) to support both queries, so that state information is not denormalized and new states are only inserted in one place. It chooses an organization for CF_1 that can support both Q_1 and Q_2, with the help of some application-side processing. Specifically, the schema partitions states according to the game server they originate from and sorts them by timestamp. This allows the extensible record store to directly support the timestamp range predicates in Q_2, and allows the application to avoid sorting by retrieving states in timestamp order. NoSE's recommended plan for Q_1 is a single `get` from CF_1 followed by application-side filtering on `PlayerID`. Similarly, Q_2's plan is a single `get` followed by filtering on `IsAdmin`. To support this filtering, NoSE has denormalized players' `IsAdmin` attributes into CF_1 to avoid the need for additional lookups to retrieve that information.

Column families CF_2 (Q_3) and CF_3 (Q_4 and Q_5) provide answers to the remaining queries, which are less frequent. Column families CF_4 and CF_5 are examples of column families that provide answers to support queries for updates, as discussed in Section 7. To insert a new player state from a given session into CF_1, the application must also know the player and server associated with that session, as well as the player's `IsAdmin` value, since that information is denormalized into CF_1 to support queries. Thus, NoSE's plan for insertions of new player states is a `get` from CF_4 to obtain the necessary player and server information for the new state's session, followed by a `put` of a new record into CF_1. Similarly, when a new session in created, NoSE's plan first obtains the `IsAdmin` value for the session's player from CF_5 before inserting the new session into CF_4.

NoSE's schema recommendations are sensitive to workload and database properties, such as the relative frequencies of the various queries and updates and the entity cardinalities. For example, CF_1 may become a poor way to support Q_1 if the number of players per game server gets too large. On the one hand, this sensitivity is a positive, as it reflects the reality of the underlying NoSQL systems, and it allows an application developer to explore the schema design space using NoSE, by simply tweaking workload parameters. On the other hand, it suggests an interesting direction for future work, which is the recommendation of schemas with performance that is robust across a range of workload changes, though not necessarily optimal at any point within the range.

9 Evaluation

In this section we present an evaluation of NoSE, designed to address two questions. First, does NoSE produce good schemas (Section 9.1)? Second, how long does it take for NoSE to generate schema recommendations (Section 9.2)? Our implementation of NoSE is available on GitHub [21].

9.1 Schema Quality

To evaluate the schemas recommended by NoSE, we used it to generate schema and plan recommendations for a target application. We then implemented the recommended schema in Cassandra along with the recommended application plans. While executing the plans against Cassandra, we measured their execution times. Similarly, we also implemented and executed the same workload against two baseline schemas for comparison.

Although extensible record stores like Cassandra are in wide use, we are not aware of open-source applications or benchmarks. One exception is YCSB [22], which is useful for performance and scalability testing of NoSQL systems, but offers no flexibility in schema design. Instead, we created a target application by adapting RUBiS [23], a Web application benchmark originally backed by a relational database which simulates an online auction website.

To adapt RUBiS for Cassandra, we created a conceptual model based on the entities managed by RUBiS. The resulting model contains seven entity sets, with ten relationships among them. Using this model, we generated a NoSE workload description, with queries and updates weighted according to the bidding workload defined by RUBiS. This workload consists of one or more statements corresponding to each SQL statement used in the original RUBiS workload.

The first schema we examine, the *NoSE schema*, was recommended by NoSE using no storage constraint. We chose not to evaluate the effect of storage constraints since updates (which we consider later) have a similar affect of reducing the amount of denormalization in the resulting schema. This results in a highly denormalized, workload-specific schema, generally consistent with the rules of thumb for NoSQL schema design discussed in Section 1. We compared this to two baseline schemas. The first, which we refer to as the *Normalized* schema, is a manually created schema which is highly normalized. For each entity set, it includes a column family for which the partition key is the primary key of the entity and which stores all data associated with the entity. The Normalized schema also includes column families which serve as secondary indices for queries which do not specify entity primary keys. These column families use the attributes given in query predicates as the partition keys and store the primary key of the corresponding entities. A human designer who is familiar with Cassandra defined the second baseline, which we refer to as the *Expert* schema, using the same workload that was input to NoSE. The expert

Q_1 Get the latest state of a player
```
SELECT states.PosX, states.PosY, states.PosZ, states.ServerTimestamp FROM
Server.sessions.states WHERE Server.ServerID = ?
AND sessions.player.PlayerID = ? ORDER BY states.ServerTimestamp
```

Q_2 Get the latest states for all players on a server
```
SELECT states.PosX, states.PosY, states.PosZ, states.ServerTimestamp,
sessions.player.PlayerID FROM Server.sessions.states WHERE
sessions.player.IsAdmin = 0 AND Server.ServerID = ? AND states.ServerTimestamp > ?
AND states.ServerTimestamp <= ? ORDER BY states.ServerTimestamp
```

Q_3 Get information on an individual server
```
SELECT Server.ServerName, Server.ServerIP FROM Server WHERE Server.ServerID = ?
```

Q_4 Check if a server exists
```
SELECT Server.ServerID FROM Server WHERE Server.ServerID = ?
```

Q_5 Get sessions for a player
```
SELECT Session.SessionID FROM Session.player WHERE player.PlayerID = ?
```

Fig. 15. Important queries in the EasyAntiCheat workload

CF_1
```
[Server.ServerID]
  [PlayerState.ServerTimestamp,
    Player.PlayerID,
    PlayerState.StateID,
    Session.SessionID][Player.IsAdmin,
      PlayerState.PosX, PlayerState.PosY,
      PlayerState.PosZ]
```

CF_2
```
[Server.ServerID][][Server.ServerName,
  Server.ServerIP]
```

CF_3
```
[Player.PlayerID][Session.SessionID][]
```

CF_4
```
[Session.SessionID][Player.PlayerID,
  Server.ServerID, Player.IsAdmin][]
```

CF_5
```
[Player.PlayerID][][Player.IsAdmin]
```

Fig. 16. Column families produced for the EasyAntiCheat workload

schema's designer also defined an execution plan for each query. The Normalized schema is 4.9GB on disk compared to 6.7GB for the expert schema and 7.8GB for the schema produced by NoSE. Details of each schema and execution plan is provided in the online supplemental materials.

We implemented each schemas in Cassandra, and populated each with data for a RUBiS instance with 200,000 users. Rather than building a custom application to target each schema, we developed a client-side execution engine which can interpret and execute the query plans specified in the plan format used by NoSE. This engine executed the plans created for all three schemas. Query and update plans for the two baselines were manually developed, and the NoSE schema uses plans recommended by NoSE.

We used two servers for each experiment, one to execute the client-side query plans and one running an instance of Cassandra 2.0.9. Each server has two six core Xeon E5-2620 processors operating at 2.10 GHz and 64 GB of memory. A 7200 RPM SATA drive stored the Cassandra data directory. The experiments ran with all Cassandra-level caching disabled, since we do not attempt to model the effects of caching in our cost model. NoSE could incorporate a more elaborate cost model which captures the effects of caching, as its cost model is pluggable.

The RUBiS workload consists of sixteen types of application-level requests, each implemented using one or more queries or updates against the underlying database. Figure 17 shows the mean response time for requests of each type, for each of the schemas that we evaluated. Mean response times for the different types ranged from 2.0–79.5 ms for the schema recommended by NoSE, 1.3–526.8 ms for the Normalized schema, and 2.7–209.4 ms for the Expert schema. The weighted overall average response times (over all request types) were 8.4ms, 87.0ms, and 41.6ms for the NoSE, Normalized, and Expert schemas, respectively. Thus, NoSE's schema results in speed-ups of $10.2\times$ and $4.9\times$ relative to the two baselines. Performance for individual requests using the NoSE schema was not better than that of the baselines for all request types. However, overall performance improves because NoSE's cost-based optimizer allows it to exploit workload information to provide good performance for the most frequent operations (those on the right in Figure 17). In particular, the NoSE schema uses extensive denormalization to support fast execution of frequent queries, at the expense of additional work during (less frequent) updates.

In addition to the experiment shown in Figure 17, we also experimented with variations of the RUBiS workload that have different mixes of request frequencies. Figure 18 shows the results, for four different mixes arranged in order of increasing write intensity. We also considered RUBiS's *Browsing* workload mix and two variations of the Bidding mix with the relative frequency of update interactions in-

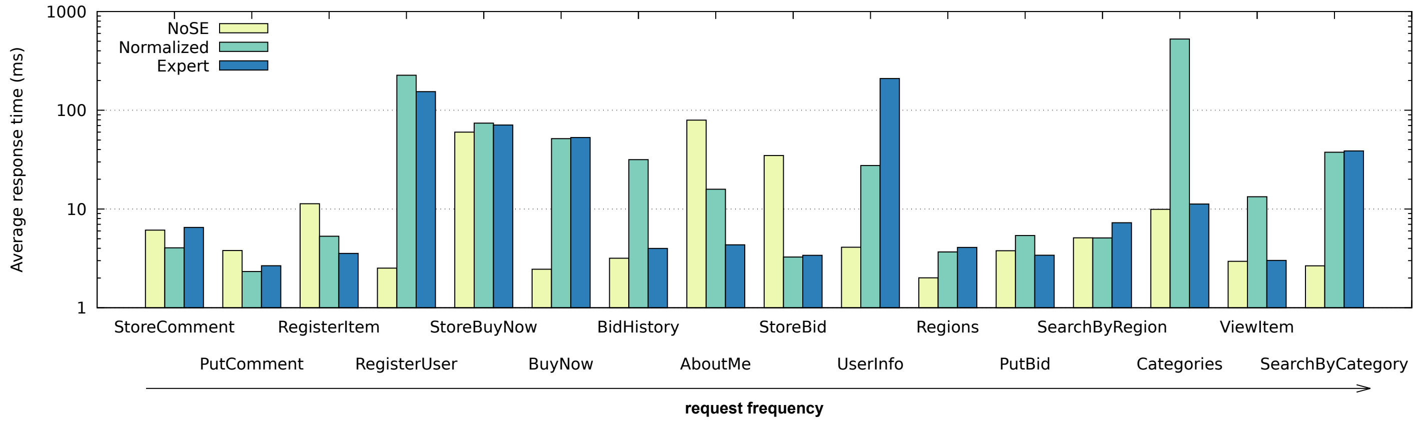


Fig. 17. Response time of RUBiS request types using three different schemas. Request types are ordered from least frequent to most frequent.

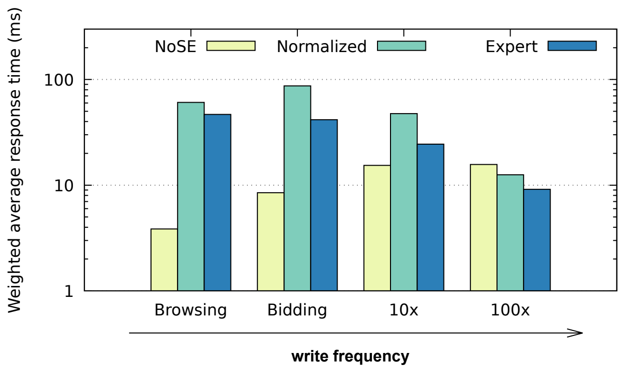


Fig. 18. Execution plan performance for different request mixes. $10\times$ and $100\times$ refer to the Bidding workload with update request frequencies increased by $10\times$ and $100\times$.

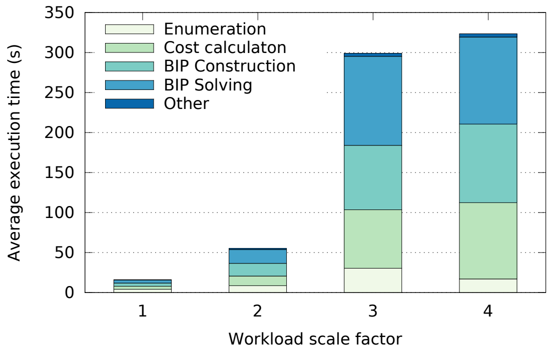


Fig. 19. Advisor runtime for varying workload scale factors

creased by factors of 10 and 100 relative to the original Bidding workload. The Browsing workload consists of 7 read-only interactions and the Bidding workload adds 9 interactions involving updates. The total frequency is approximately 23% update interactions for the Bidding workload. For each mix, NoSE generated a schema and execution plans specific to that mix, which we compared against the original Expert and Normalized schemas. Note that we only expect the Expert schema to perform well against the Bidding workload used for its development.

NoSE is extremely effective on the Browsing mix, largely because it is free to denormalize heavily, with no update penalty. As the workload becomes more write intensive, NoSE's schema recommendations become more normalized, and workload performance approaches that achieved by the Normalized baseline. In the most update intensive mix, NoSE's schema performs slightly worse than the baseline schemas. NoSE has no knowledge of the correlation of queries in the input workload and cannot share the results of support query execution. In contrast, the expert schema does exploit this knowledge and is thus able to avoid unnecessary queries.

9.2 Advisor Runtime

Running NoSE for the RUBiS workload takes approximately 3.3 minutes. To evaluate the advisor runtime for workloads larger than RUBiS, we generated random entity graphs and queries to use as input to our tool. The entity graph generation uses the Watts-Strogatz random graph model [24]. After generating the graph, we randomly assign a direction to each edge and create a foreign key at the head node. We then add a random number of attributes to each entity in the graph. Our generator uses a random walk through the graph to identify the graph of each statement. For any statements involving a `WHERE` clause, we randomly generate predicates in the graph. Queries and updates select or update randomly chosen attributes in the graph.

Figure 19 shows the results of a simple experiment in which we started with a random workload having similar properties to the RUBiS workload discussed in the previous section. We then increased the size of the workload by multiplying the number of entities and statements by a constant factor. The largest workload ($4\times$ scale factor)tested contains 120 queries, 12 updates, and 20 insertions over an entity graph with 28 entities. The figure shows the time required for NoSE to recommend a schema and a set of execution plans as a function of this factor. We ran all experiments using a machine with the same specifications as in the previous section. The increase in runtime is a result of the increased number column families enumerated, which also increases the number of support queries NoSE considers. This interaction increases non-linearly with the workload size (e.g., increased numbers of queries and updates) since there are more ways that column families recommended for queries interact with updates. There is likely room for optimization in the NoSE code to significantly reduce the runtime. For example, any heuristics which can exclude column families from enumeration will reduce the runtime at all further stages in the process.

10 Related Work

Numerous tools are available for solving related design problems in relational database systems. Many of these tools select an optimized collection of indexes and materialized views to support a given workload [9], [10], [11], [12], [13], [14], [15], [16]. However, as noted in Section 2, there are some significant differences between relational physical schema design and schema design for NoSQL systems, which NoSE addresses. Others focus on vertical partitioning of relations, either to recommend a set of covering indexes [25] or to determine a physical representation for the relations [26], [27]. There are also tools for automating relational partitioning and layout across servers [11], [28] or storage devices [29], [30]. DBDesigner [31] determines the physical representations (called projections) of tables for the Vertica [26] column store, based on an input workload. However, DBDesigner recommends a single projection at each iteration which may not produce a globally optimal solution. In addition, DBDesigner does not explicitly consider the effect of updates but instead relies on heuristics to limit the number of projections.

Typically, relational physical design involves the identification of candidate physical structures, followed by the selection of a good subset of these candidates. NoSE uses the same general approach. Some relational design tools, including CoPhy [14], [16], CORADD [15], and a physical design technique for C-Store [27] have formulated the task of choosing a good set of candidates as a BIP. As noted in Section 5, Papadomanolakis and Ailamaki [14] presented a simple formulation of the problem as a binary integer program. CoPhy [16], an extension of this work, adds optimizations to reduce calls to the relational query optimizer. As in our work, their approach exploits the decomposition of queries into components and analyzes these components independently. CoPhy also includes a rich set of constraints which may be useful as extensions to NoSE.

Our approach to the schema design problem for extensible record stores owes an intellectual debt to GMAP [32], which is a technique for improving the physical data independence of relational database systems. In GMAP, the authors describe both application queries and physical structures using a conceptual entity-relationship model. GMAP associates one or more physical structures with each query which can provide an answer to the query. The authors use this approach out of a desire to provide a more thorough form of physical data independence. In our case, we adopt a similar approach out of necessity, as the extensible record stores we target do not implement separate logical and physical schemas. However, in GMAP, the primary algorithmic task is to map each query to a given set of physical structures. In contrast, our task is to *choose* a set of physical structures to handle a given workload, in addition to specifying which physical structures provide an answer to each query.

Others have also proposed writing queries directly against a conceptual model. For example, ERQL [33], is a conceptual query language over enhanced ER diagrams. It defines *path expressions* referring to a series of entities or relationships. Our query model is somewhat more restrictive as we disallow self references. Queries over our conceptual model are also similar to path expressions in object databases, and the physical structures our technique recommends are similar to the nested indexes and path indexes described by Bertino and Kim [34].

Vajk et al. [35] discuss schema design in a setting similar to ours. Their approach, like ours, starts with a conceptual model with queries expressed in the UML Object Constraint Language. They sketch an algorithm that appears to involve the use of foreign key constraints in the conceptual model to exhaustively enumerate candidate denormalizations. An unspecified technique is then used to make a cost-based selection of candidates. Although this approach is similar to ours, it is difficult to make specific comparisons because the schema design approach is only sketched. Rule-based approaches also exist for adapting relational [36] and OLAP [37] schemas for NoSQL databases. However, these approaches are workload agnostic and do not necessarily produce schemas which can efficiently implement any particular workload. Li [36] suggests a workload-aware approach such as the one we take with NoSE as future work.

11 Conclusion

Schema design for NoSQL databases is a complex problem with additional challenges as compared to the analogous problem for relational databases. We have developed a workload-driven approach for schema design for extensible record stores, which is able to effectively explore trade-offs in the design space. Our approach implicitly captures best practices in NoSQL schema design without relying on general design rules-of-thumb, and is thereby able to generate effective NoSQL schema designs. Our approach also allows applications to explicitly control the tradeoff between normalization and query performance by varying a space constraint.

Currently, NoSE only targets Cassandra. However, we believe that with minimal effort, the same approach could apply to other extensible record stores, such as HBase. We also intend to explore the use of a similar model on data stores with different data models, such as key-value stores and document stores. Applying our approach to similar data stores may only require changing the cost model and the physical representation of column families. However, we imagine NoSE may require more significant changes to fully exploit the capabilities of different data models.

Acknowledgements

Thanks to EasyAntiCheat for their collaboration in sharing knowledge of their workload. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Also thanks to Cătălin Avram for assisting with manual schema design.

References

[1] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

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

[3] "HBase: A distributed database for large datasets," retrieved March 7, 2013 from https://hbase.apache.org.

[4] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006.

[5] J. Patel, "Cassandra data modeling best practices, part 1," 2012, retrieved Oct. 21, 2014 from http://ebaytechblog.com/?p=1308.

[6] N. Korla, "Cassandra data modeling - practical considerations @ Netflix," 2013, retrieved Oct. 21, 2014 from http://www.slideshare.net/nkorla1share/cass-summit-3.

[7] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu, "NoSE: Schema design for NoSQL applications," in *ICDE*, 2016, pp. 181–192.

[8] E. Hewitt, *Cassandra: The Definitive Guide*, 2nd ed., M. Loukides, Ed. Sebastopol, CA: O'Reilly Media, 2011.

[9] S. J. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Transactions on Database Systems*, vol. 13, no. 1, pp. 91–128, 1988.

[10] S. Agrawal *et al.*, "Automated selection of materialized views and indexes in SQL databases," *PVLDB*, pp. 496–505, 2000.

[11] D. C. Zilio *et al.*, "DB2 design advisor: integrated automatic physical database design," *PVLDB*, pp. 1087–1097, 2004.

[12] B. Dageville *et al.*, "Automatic SQL tuning in Oracle 10g," *PVLDB*, vol. 30, pp. 1098–1109, 2004.

[13] N. Bruno and S. Chaudhuri, "Automatic physical database tuning: A relaxation-based approach," in *SIGMOD*, 2005.

[14] S. Papadomanolakis and A. Ailamaki, "An integer linear programming approach to database design," *ICDEW*, pp. 442–449, 2007.

[15] H. Kimura *et al.*, "CORADD: Correlation aware database designer for materialized views and indexes," *PVLDB*, pp. 1103–1113, 2010.

[16] D. Dash, N. Polyzotis, and A. Ailamaki, "CoPhy: A scalable, portable, and interactive index advisor for large workloads," vol. 4, no. 6, pp. 362–372, 2011.

[17] P. P.-S. Chen, "The entity-relationship model - toward a unified view of data," *ACM TODS*, vol. 1, no. 1, pp. 9–36, 1976.

[18] DataStax, "About indexes in Cassandra," 2014, retrieved Nov. 4, 2014 from http://docs.datastax.com/en/archived/cassandra/1.1/docs/ddl/indexes.html.

[19] Gurobi Optimization, Inc., "Gurobi optimizer reference manual," 2014, retrieved Aug. 8, 2014 from https://www.gurobi.com.

[20] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Comput. Surv.*, vol. 24, no. 1, pp. 63–113, Mar. 1992.

[21] M. J. Mior, "NoSE - automated schema design for NoSQL applications," 2016, retrieved Jan. 15, 2016 from https://github.com/michaelmior/NoSE.

[22] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. on Cloud Computing*, 2010.

[23] E. Cecchet *et al.*, "Performance and scalability of EJB applications," *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 246–261, 2002.

[24] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[25] S. Papadomanolakis and A. Ailamaki, "AutoPart: Automating schema design for large scientific databases using data partitioning," in *SSDBM*, 2004, pp. 383–392.

[26] A. Lamb *et al.*, "The Vertica analytic database: C-Store 7 years later," *PVLDB*, vol. 5, no. 12, pp. 1790–1801, 2012.

[27] A. Rasin and S. Zdonik, "An automatic physical design tool for clustered column-stores," *EDBT*, pp. 203–214, 2013.

[28] J. Rao *et al.*, "Automating physical database design in a parallel database," in *SIGMOD*, 2002, pp. 558–569.

[29] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya, "Automating layout of relational databases," in *ICDE*, 2003, pp. 607–618.

[30] O. Ozmen *et al.*, "Workload-aware storage layout for database systems," in *SIGMOD*, 2010, pp. 939–950.

[31] R. Varadarajan *et al.*, "DBDesigner: A customizable physical design tool for Vertica analytic database," in *ICDE 2014*, 2014, pp. 1084–1095.

[32] O. G. Tsatalos *et al.*, "The GMAP: a versatile tool for physical data independence," *PVLDB*, vol. 5, no. 2, pp. 101–118, 1996.

[33] M. Lawley and R. Topor, "A query language for EER schemas," in *ADC 94*, 1994, pp. 292–304.

[34] E. Bertino and W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 196–214, 1989.

[35] T. Vajk, L. Deák, K. Fekete, and G. Mezei, "Automatic NoSQL schema development: A case study," in *Artificial Intelligence and Applications*. Actapress, 2013, pp. 656–663.

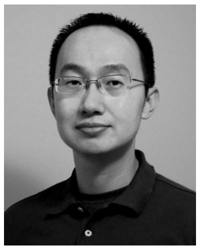
[36] C. Li, "Transforming relational database into HBase: A case study," in *Proc. ICSESS 2010*, 2010, pp. 683–687.

[37] M. Chevalier *et al.*, "Implantation not only SQL des bases de données multidimensionnelles," in *Colloque VSST*. VSST, 2015.

Michael J. Mior received an MSc degree from the University of Toronto in 2011. He is currently a PhD student at the University of Waterloo, Canada. His research interests include designing schemas for NoSQL databases. He is a student member of the IEEE.

Kenneth Salem is a professor in the Cheriton School of Computer Science at the University of Waterloo, where he has been since 1994. He received his PhD in computer science from Princeton University, and spent several years at the University of Maryland before moving to Waterloo. He has also held visiting positions at IBM's Almaden Research Center and at HP Labs in Palo Alto. His research interests span a variety of topics related to database management, storage systems, and cloud computing.

Ashraf Aboulnaga is a Research Director at the Qatar Computing Research Institute and an Adjunct Associate Professor position at the University of Waterloo. His current research interests include platforms for Big Data, distributed storage systems, and data integration. Aboulnaga obtained MS and PhD degrees in computer science from the University of Wisconsin - Madison. He received a Google Research Award, the Ontario Early Researcher Award, and Best Paper Awards at the VLDB 2011 and SoCC 2015 conferences. He currently serves on the editorial boards of the IEEE Transactions on Knowledge and Data Engineering, the VLDB Journal, and Distributed and Parallel Databases. He is a Senior Member of the IEEE and an ACM Distinguished Scientist.



Rui Liu is a system software engineer at HPE Vertica. His research interests include database systems, analytics infrastructure, and cloud computing. Rui received a PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences. He has been a PC member at DEXA Conferences.