NoSE: Schema Design for NoSQL Applications

Michael J. Mior, Kenneth Salem
University of Waterloo
{mmior,kmsalem}@uwaterloo.ca

Ashraf Aboulnaga
Qatar Computing Research Institute, HBKU
aaboulnaga@qf.org.qa

Rui Liu
HP Vertica
r.liu@hp.com

***Abstract*—Database design is critical for high performance in relational databases and many 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 for this 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.

I. 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 one particular type of NoSQL system, termed *extensible record stores* in Cattell's taxonomy [1]. In these systems, applications can create tables of records, with each record identified by a key. However, the set of columns, in the records need not be defined 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 an extensible record store application can be developed, 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 specific columns need not be defined in advance, it is still necessary to decide what column families will exist in the record store, and what information will be encoded in each column family. 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, current practices in schema design for extensible record stores are captured in 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 relational tables, ensuring that column families reflect the anticipated workload, and denormalizing data in the record store. While such recommendations are clearly useful, they are necessarily vague and generic, and must be tailored to each application. In Section II, we illustrate some of the choices faced when designing a schema for a specific 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 extensible record store schema optimized for a target application. Our tool uses a cost-based approach to schema optimization. By estimating the performance that candidate schemas would have for the target application, we recommend the schema that results in the best estimated performance. NoSE is intended to be used 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, NoSE also recommends how the application should be implemented against the schema.

This paper makes the following contributions. First, we formulate the *schema design problem* for extensible record stores. Second, we propose a cost-based solution to the schema design problem, which is embodied in a schema design advisor we call *NoSE*, the *NoSQL Schema Evaluator*. Using a conceptual model of the data required by a target application, as well as a description of how that data will be used, NoSE recommends both an extensible record store schema, i.e., a set of column family definitions which is optimized for the target application, and guidelines for application development using this schema. Finally, we present an evaluation of this approach. For the evaluation, we use a simple online auction Web application as the target and we evaluate both the quality of the NoSE-recommended schema and the time required to generate recommendations.

II. 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 data that will be managed by this application are described by the conceptual model in Figure 1, adapted from Hewitt [7].

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

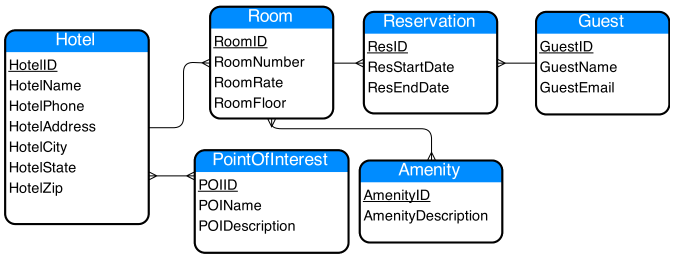


Fig. 1. Entity graph for a hotel booking system. Each box represents an entity set, and edges between boxes represent relationships.

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 that have been 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, this query could be answered easily if the record store includes 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 would include one column for each POI that is 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, columns may be added to or removed from a guest's record when that guest's hotel bookings are updated 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 which attributes' values will be used as record keys in the column family. The second element indicates which attributes' values will be used as column names, and the third indicates which will be used as column values. We refer 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.

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 the point of interest information is denormalized: the name and description of a POI may be stored in many guests' records. 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.

The schema design problem that NoSE solves is related to schema design for relational databases. However, there are also significant differences between the two. Relational systems provide a clean separation between logical and physical schemas. Standard procedures exist for translating a conceptual model, like the entity graph in Figure 1 to a normalized logical relational schema, i.e., a set of table definitions, against which the application's workload can be defined. The logical schema often determines a physical schema consisting of a set of base tables. The physical layout of these base tables can then be optimzied and they can then be 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 [8]–[15].

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 the entire workload to be implemented. We provide further discussion of relational physical design tools in Section VIII.

III. System Overview

Figure 2 gives a high level illustration of the NoSE schema advisor. NoSE is intended to be used early in the process of developing an application using an extensible record store backend. It takes two inputs. The first is a *conceptual model* of the data required by the application. The second is a description of the *workload*, indicating how the application expects to query and update the record store.

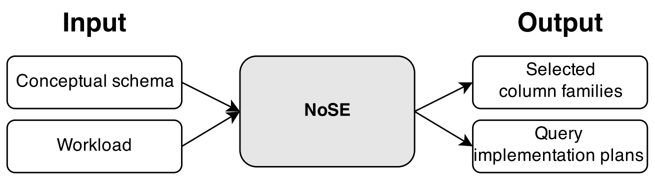


Fig. 2. Schema advisor overview

Given these inputs, the advisor produces two outputs. The first is a recommended *schema*, which describes the column families that should be 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 are used as a guide for the application developer.

In the remainder of this section, we describe the conceptual model, workload, and schema of the record store in more detail. Query plans are described in more detail in Section IV-C.

A. Database Conceptual Model

To recommend a schema for the target application, NoSE must have a conceptual model describing the information that will be stored 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 [16]. 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, one or more of which serve as a key. For example, the model shown in Figure 1 indicates that each room has an associated room number and rate. In addition, each room is associated with a hotel and with a set of reservations.

B. Workload Description

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

Each query in the workload returns information about a particular type of entity. 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 in given city at a given minimum rate. In this example, `?city` and `?rate` are parameters, values for which would be provide by the application when it performs such a query. NoSE queries are expressed directly over the conceptual model. Specifically, each query identifies a *target entity set* (in the `FROM` clause) and a path that originates at the target entity set and traverses the entity graph. Each query returns the attribute values from one or more entities along this path.

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 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.

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

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

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

C. 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. Records in a single partition are ordered by the clustering keys. For example, in Section II, 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` and `get` 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 assocaited with a given `GuestID`. Similarly, a `put` operation can be used to modify the $\mathcal{C} \mapsto \mathcal{V}$ pairs associated with a single partition key.

Some extensible record stores provide additional capabilities beyond the basic `get` and `put` 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

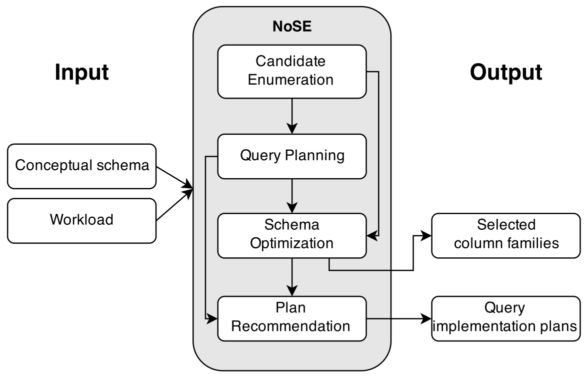


Fig. 4. Complete schema advisor architecture

a limited form of secondary indexing, allowing applications to select records by something other than the partitioning key. However, many applications do not use them for performance reasons [17]. For simplicity, we restrict ourselves to the simple `get`/`put` model we have described, as it captures functionality that is commonly present in extensible record stores.

D. The Schema Design Problem

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

Given a conceptual schema, 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 can be answered 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 IV-C.

IV. Schema Advisor

Given an application's conceptual schema and workload, as shown in Figure 2, our advisor proceeds through four steps to produce a recommended schema and a set of query implementation plans.

1) **Candidate Enumeration** The first step is to 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** The advisor generates 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** The possible plans for the queries are used to generate a binary integer program (BIP) used to choose a subset of the candidate column families to form the recommended schema. The BIP is then given to an off-the-shelf solver (we have chosen to use Gurobi [18]) which chooses a set of column families that minimizes the cost of answering the queries.
4) **Plan Recommendation** The advisor chooses 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.

This process is illustrated in Figure 4. In the reminder of this section, we discuss candidate enumeration and query planning. Schema optimization and plan recommendation are presented in Section V.

A. 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.

Instead, we enumerate candidates using a two-step process based on the application's workload. First, we enumerate candidate column families for each query in the application workload using an algorithm $\texttt{Enumerate}(q)$. The union of these sets C is used as our initial pool, i.e. $C \leftarrow \bigcup_q \texttt{Enumerate}(q)$. Second, we supplement this pool with additional column families constructed by combining candidates from the initial pool, $C \leftarrow C \bigcup \texttt{Combine}(C)$. 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. Both the `Enumerate` and `Combine` algorithms are described in the following sections. We note that we do not claim to enumerate column families which result in an optimal schema. Any enumeration algorithm which produces valid column families capable of answering queries in the workload could be substituted here. We leave heuristics to determine additional useful column families as future work. However, the optimization process we discuss in Section V produces the optimal subset of the enumerated candidates for the given cost model.

1) Candidate Column Families: Recall from Section III-C 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. For example, using pairs `(HotelCity, HotelState)` as partition keys, `HotelNames` as clustering keys, and `(HotelAddress, HotelPhone)` as values, we can define a column family that can be used to retrieve, for a given city and state, a list of hotel names, addresses, and phone numbers, in order of hotel name. 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. Each column family has an associated path of relationships linking the entities contained in the column family. However, for the examples we show, the path is unambiguous and is omitted from the description. For example, the column family described above would be represented as

```
[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 that can be used to directly retrieve answers to that query, which we call a *materialized view*. 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, corresponds to the following column family:

```
[HotelCity][RoomRate, GuestID]
  [GuestName, GuestEmail]
```

By supplying a city name and a minimum room rate, an application can use this column family to retrieve a list of (`RoomRate`, `GuestID`) pairs, with each mapped to the name and email address of the specified guest. WE do not show it here, but we also include the ID of each entity along the path in the clustering key (e.g. `HotelID`, `RoomID`, and `ResID`). This ensures we have a unique record for each guest reservation since the same guest and hotel may be connected multiple ways (e.g. through different reservations for the same hotel). The use of `HotelCity` in the partition key restricts the results to a given city while having `RoomRate` as part of the clustering key allows for range queries to find reservations over the given rate.

2) Per-Query Candidate Enumeration: Since each query corresponds to a column family, the `Enumerate` algorithm could enumerate just one candidate for each query, namely a column family which has the appropriate attributes to answer the query using a single `get` operation against the extensible record store. This single column family is the materialized view for the query as described above. However, the schema optimizer requires more flexibility than this, 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, `Enumerate` also includes additional column families in the candidate pool for each query. Each provides a partial answer for the 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 entity along the query path. Decomposing a query at a specific entity along its path splits the query into two parts, which we call the *prefix query* and a *remainder query*. `Enumerate` produces one or more candidate column families based on the prefix query, then recursively decomposes the remainder query using the same approach. Figure 5 illustrates the first level of this recursive splitting process for the example query that was shown in Figure 3. The query path for this query is `Guest.Reservation.Room.Hotel`, so the query is decomposed at four points.

For each prefix query, NoSE enumerates a column family with the appropriate construction to allow the prefix query to be answered with a single request to the backend record store. In addition, depending on the form of the prefix query, the enumerator may enumerate additional candidates. If the `SELECT` clause of the prefix query includes attributes not in the key of the target entity, two additional candidates will be enumerated: one that returns only the key attributes, and a second that returns the attributes from the `SELECT` clause, given a key. For example, for the first prefix query in Figure 5, in addition to the materialized view for that query, which is

```
[HotelCity][RoomRate, GuestID]
  [GuestName, GuestEmail]
```

we also enumerate the following two column families:

```
[HotelCity][RoomRate, GuestID][]
[GuestID][][GuestName, GuestEmail].
```

The former can be used to return a set of `GuestID`s, given a city and a room rate, and the latter can then be used to retrieve the guests' names and email addresses.

Finally, when the prefix query contains multiple predicates, the enumerator generates additional candidates corresponding to *relaxed* versions of the prefix query. Specifically, when the enumerator considers a query of the form

```
SELECT attribute-list FROM entity WHERE
entity.attr op ? AND predicate2 AND ...
```

it also generates materialized view column families for relaxed queries of the form

```
SELECT attribute-list, attr FROM entity
WHERE predicate2 AND ...
```

That is, we remove one or more predicates and add the attributes involved in the predicates to the `SELECT` list. These column families can be used to retrieve a superset of the result of the original prefix query, which can then by filtered by the application. When relaxing prefix queries, the enumerator removes only predicates that test an attribute of the target entity (the one mentioned in the `FROM` clause). Furthermore, predicates are only considered for removal if the remaining query will have at least one equality predicate remaining. (This is required to construct a valid `get` request on the column family that will be constructed by the schema advisor.)

We can relax queries involving ordering in the same way by moving an attribute in an `ORDER BY` clause to the `SELECT` list. The plan for a query using this column family would then perform a sort in the application on the selected attribute.

3) Candidate Combinations: Once candidates have been enumerated for each query in the workload, `Combine` considers additional candidates by combining the per-query candidates that are already present in the pool. There are many opportunities for creating new column families by combining candidates from the pool, but `Combine` currently only exploits one simple opportunity. Specifically, `Combine` looks for pairs of column families in the pool for which

Decomposition Point	Prefix query	Remainder query
`Guest`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.Room.Hotel.HotelCity = ? AND Guest.Reservation.Room.RoomRate > ?`	none
`Reservation`	`SELECT Reservation.ResID FROM Reservation WHERE Reservation.Room.Hotel.HotelCity = ? AND Reservation.Room.RoomRate > ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.ResID = ?`
`Room`	`SELECT Room.RoomID FROM Room WHERE Room.Hotel.HotelCity = ? AND Room.RoomRate > ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.Room.RoomID = ?`
`Hotel`	`SELECT Hotel.HotelID FROM Hotel WHERE Hotel.City = ?`	`SELECT Guest.GuestName, Guest.GuestEmail FROM Guest WHERE Guest.Reservation.Room.Hotel.HotelID = ? AND Guest.Reservation.Room.RoomRate > ?`

Fig. 5. Example of query decomposition for candidate enumeration

- both column families have the same partition key, and
- both column families have no clustering key, and
- the two column families have different data attributes.

For each such pair, `Combine` adds an additional column family to the candidate pool. The new column family has the same partition keys as the column families on which it is based, and the union of their value attributes. Thus, the new candidate will be larger than either of the original candidates but will be potentially useful for answering more than one query.

Increasing the number of candidates increases the opportunity for the schema advisor to identify a high-quality schema but 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.

B. *Application Model*

To determine which candidate column families to recommend, the schema advisor must understand the way in which the candidates will be used by the application. This understanding is based on an *application model*. This model is used in two ways. First, it allows the advisor to estimate the cost (to the application) of executing queries in the workload under a particular schema recommendation by using the model to determine how the application would use the recommended schema to obtain answers to the workload queries. Second, the schema advisor can use the model as a framework for describing to the application developers how the recommended schema should be used to answer the queries. We refer to these descriptions as *query implementation plans* or simply, plans. The schema advisor's output includes a plan for each application query in the input workload. We note that the exact cost model used to estimate the cost of each query implementation plan is not important to our approach. The simple cost model we have developed is detailed in B.

Consider the example query from Figure 3, and suppose the following column families exist in the schema:

CF_1: [`HotelCity`][`RoomID`][`RoomRate`]
CF_2: [`RoomID`][`GuestID`][`GuestName`, `GuestEmail`]

The application can use these column families to answer the example query by performing the following steps:

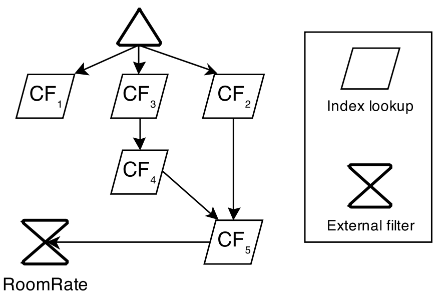
1) Using the given city as the partition key value, perform a `get` operation on CF_1 to obtain a list of `RoomID`, `RoomRate` pairs.
2) Using the given minimum room rate, filter the list of `RoomID`, `RoomRate` pairs to eliminate those with a `RoomRate` less than the minimum allowable rate.
3) For each remaining `RoomID`, use the `RoomID` as the partition key value to perform a `get` on CF_2 to obtain a list of `GuestID`, `GuestName`, `GuestEmail` triples. Merge the results (discarding duplicates) to obtain the final query result.

We consider this plan to consist of instances of four primitive operation types, which form part of the application model. The first is `get` operations on the underlying record store. The second is a *filter* operation, which is used in step 2 to eliminate rooms with low rates from the result of the `get` operation in step 1. The last is a *join* operation, which is used in step 3 to take results from an earlier step and use them as keys to `get` data from another column family. Finally, our application model also includes a *sort* operation, which may be used, for example, when the schema does not allow the application to retrieve results in the desired order. Of these operations, only data access (the `get` operation) is implemented by the NoSQL data store. Filtering, sorting, and joining, when required, must be performed on the client side by the application itself.

C. *Query Planning*

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 application operations, 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 of the candidate column families to recommend.

Query planning in our advisor is performed as part of the same recursive decomposition process that is used to generate candidate column families. Consider the decomposition of the



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

Fig. 6. Example query plan space

running example query (Figure 3) that was shown in Figure 5. 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 of the candidate column families that was generated for that prefix query.

Figure 6 shows the plan space for the simple relaxed prefix query seen in Section IV-A.

```
SELECT Room.RoomID FROM Room WHERE
Room.Hotel.HotelCity = ?city 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 `RoomID`s for the given hotel using CF_4. Finally, the `RoomRate`s are discovered using CF_5 and the `RoomID`s are filtered to only contain those matching the predicated on `RoomRate`. The final plan is similar, but goes directly from a `HotelCity` to a list of `RoomID`s using CF_2.

Note that the number of column families that can be used to execute these plans is larger than what is shown. Each plan can make use of any column family which contains the necessary data, but possibly "larger" through the addition of some suffix to the clustering key or additional data.

V. 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 [13] 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

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

subject to

All used column families being present

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

Maximum space usage S

$$\sum_j s_j \delta_j \leq S$$

Plus per-query path constraints (see text)

Fig. 7. Binary integer program for schema optimization

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 as in 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 relatively 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 IV-C to generate a BIP. The program uses a binary decision variable, δ_{ij}, 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 $\sum_i \sum_j C_{ij}\delta_{ij}$, where C_{ij} represents the cost of using the jth column family in the plan for the ith query. However, after solving this optimization problem, we run the solver a second with an additional constraint that the cost of the workload equals the minimum value which was just discovered. The new objective function is the total number of column families in the recommended schema. This allows NoSE to produce the smallest schema out of 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 corresponding column families will be included in the set of column families 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 jth column family is included in the set of recommended column families if it is used 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.

Overall, this approach requires $|Q||P|$ variables representing the use of column families in query implementation plans, and

$|P|$ variables representing which column families should be recommended where $|Q|$ represents the number of queries and $|P|$ the number of candidate column families. 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. The binary integer program is summarized in Figure 7.

As noted in Figure 7, the BIP also requires a set of *path 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 are derived from the per-query plan spaces determined by the query planner. For example, in Figure 6, at most one of CF$_1$, CF$_3$, and CF$_2$ can be selected for use in answering this query, since each is useful for different plans, and only one plan is selected per query. In addition, if CF$_3$ is selected for use, then CF$_4$ must also be selected. The BIP will include corresponding constraints on the decision variables δ_{ij} that indicate whether those column families are used to answer this query. For this example, the following constraints would be generated for the example in Figure 6 (assuming we number the query as query 1):

$$\begin{aligned} \delta_{1,1} + \delta_{1,3} + \delta_{1,2} &= 1 \\ \delta_{1,4} &= \delta_{1,3} \\ \delta_{1,5} &\geq \delta_{1,4} + \delta_{1,2} \end{aligned}$$

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.

VI. 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 which contains an attribute that is modified by an update must also be modified, so repetition of attributes increases update cost.

We first introduce extensions to our workload description to express updates. We then discuss how NoSE executes these updates for a given set of column families. Finally, we describe modifications required to the enumeration algorithm and the BIP used by NoSE in order to support these updates. Extensions to consider such tasks as performing aggregation at insertion time are left as future work.

A. Update Language

In order to support updates to the workload, we extend our query language with additional statements which support updates to data according to the conceptual model as in the examples shown in Figure 8. `INSERT` statements create new entities and result in insertions to column families containing attributes from this entity. We assume that the primary key of each entity is provided with the entity when it is inserted, but all other attributes are optional. `UPDATE` statements modify

```
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. 8. Example update statements in the workload

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

Materialized view
```
[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. 9. An example workload against the hotel schema

attributes resulting in updates to any corresponding column families in the schema. `DELETE` statements remove entities and all data about that entity is removed 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 allow for relationships between entities to be created or removed. These statements simply specify the primary key of each entity and the relationship to modify. We also allow relationships to be created when a new entity is inserted.

B. Update Execution

As with queries, NoSE must provide an implementation plan for each update. To update a column family, we delete records for old data which is being modified or removed (in the case of `UPDATE`, `DELETE`, and `DISCONNECT`). We then insert records corresponding to the new data (in the case of `UPDATE`, `INSERT`, and `CONNECT`). For the example given in Figure 9, we would update the materialized view by first removing the record for the old `RoomRate`. This would be a tuple consisting of values for `RoomFloor`, `POIID`, `HotelID`, and `RoomID`. The associated `RoomRate` is not required since the partition and clustering key attributes constitute the primary key for the record.

In general, to modify records we must have the primary key (partition and clustering key attributes) available for each record in each column family which needs to be modified in order to construct a valid `put` request. Since these attributes may not be provided with the update, we construct one or more *support queries* which select values for the attributes in the primary key primary key given the predicates specified in the update. Support query construction varies depending on the type of update, with details provided in A. A complete plan

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

subject to

All constraints from Figure 7

Additional constraints per support query (see text)

Fig. 10. BIP modifications for updates

for an update consists of executing support queries followed by insertion and/or deletion into the associated column family.

C. Column Family Enumeration for Updates

Additional column families may be necessary to answer support queries for updates in the workload. To expand our enumeration to include updates, we use the modified procedure shown in Algorithm 1. The `Enumerate` and `Combine` functions are the same as previously described in Section IV-A. The `Modifies?` function is a predicate which tests if an update requires modifications to a given column family. Finally, the `Support` function produces all the support queries necessary for a given update on a column family. Note that we run the candidate enumeration process twice for support queries. This is because when generating support queries for statements in the original workload, we may cover new paths in the entity graph. As a result, it is possible that there are no column families suggested by `Enumerate` which can provide answers to these support queries. By performing enumeration for these support queries a second time, we are guaranteed to cover all paths which were expanded in the first step.

```
input : A set of queries $Q$ and updates $U$
output: A set of enumerated column families
/* enumeration for workload queries */
$C \leftarrow \emptyset$;
foreach query $q$ in $Q$ do
    $C \leftarrow C \bigcup$ Enumerate($q$);
end
/* enumeration for support queries */
do twice
    $C' \leftarrow C$;
    foreach update $u$ in $U$ do
        foreach column family $c$ in $C'$ do
            if Modifies?($u$, $c$) then
                foreach query $q$ in Support($u$, $c$) do
                    $C \leftarrow C \bigcup$ Enumerate($q$);
                end
            end
        end
    end
end
return $C \bigcup$ Combine($C$);
```

Algorithm 1: Column family enumeration for updates

D. BIP Modifications

To incorporate updates into our BIP, we first add constraints for all support queries in a similar manner to queries in the original workload. However, there is an additional constraint that ensures a plan is only generated for a support query if the decision variable for the associated column family is set. This ensures that we do not generate implementation plans for support queries which do not need to be executed. $\sum_m \sum_n C'_{mn}\delta_n$ is added to the objective function to represent the cost of updating each column family which is contingent on the column family being selected for the final schema. C'_{mn} is the cost of updating column family n for update m given that the column family is selected for inclusion in the final schema (δ_n). The cost support queries is also added using the same weight specified for the update in the workload. This modification to the objective function constrains the problem to reduce denormalization of attributes which are heavily updated. These modifications are shown in Figure 10.

After solving this new BIP, we plan 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. Note that we only need to generate update plans for column families which are included in the recommended schema (i.e. those that have their decision variables set).

VII. EVALUATION

In this section we present an evaluation of NoSE, designed to address two questions. First, does NoSE produce good schemas (Section VII-A)? Second, how long does it take for NoSE to generate schema recommendations (Section VII-B)?

A. 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 the record store, we measured their execution times. In a similar manner, we also implemented and executed the same statements against two other schemas for comparison, which we describe later. Our implementation of NoSE is available on GitHub [19].

Although extensible record stores like Cassandra are in wide use, we are not aware of open-source applications or benchmarks, with the exception of YCSB [20], which is intended for performance and scalability testing of NoSQL systems, but offers no flexibility in schema design. Instead, we created a target application by adapting RUBiS [21], a Web application benchmark originally backed by a relational database which simulates an online auction site.

To adapt RUBiS for Cassandra, we created a conceptual model based on the entities managed by RUBiS. The resulting model contains eight entity sets, with eleven relationships among them. Using this model, we generated a workload weighted according to the original RUBiS request distribution. This workload consists of one or more statements corresponding to each SQL statement used by the RUBiS bidding workload. We excluded RUBiS queries which involve browsing and searching by region as they cannot be efficiently implemented using our current model.

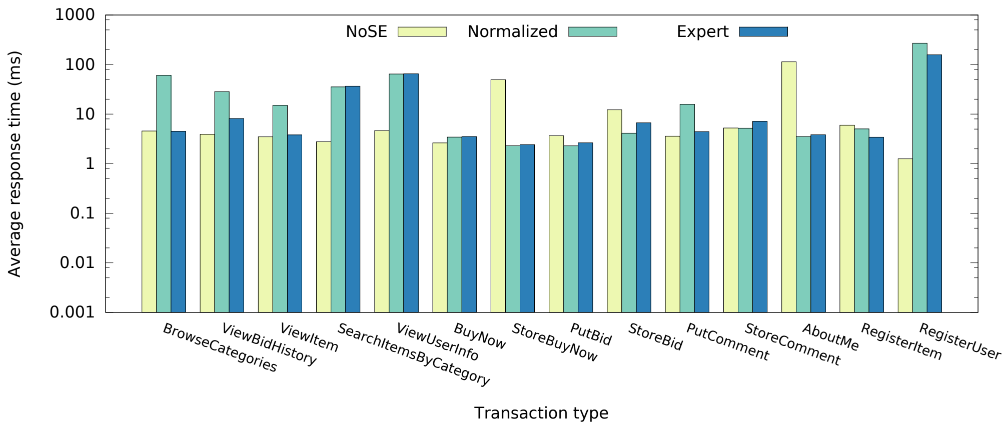


Fig. 11. Bidding workload performance on different schemas

The first schema we examine, the *recommended schema*, is determined by the NoSE advisor with no storage constraint. This results in a highly denormalized, workload-specific schema, generally consistent with the rules of thumb for NoSQL schema design discussed in Section I. Second, the *normalized schema* is a manually created schema which is highly normalized. This schema contains a column family for each entity set where the partition key is the primary key of the entity and containing all data associated with the entity. The schema also contains 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. Finally, we defined a third schema which we refer to as the *"expert" schema*. This schema was defined manually by a human designer familiar with Cassandra using the same workload input to NoSE. In addition to this schema, the designer also provided a set of execution plans for each query. A detailed description of each schema is available in C.

We used each schema to define a set of column families in Cassandra, and populated the record store using data for a RUBiS instance with 200,000 users. Finally, we developed a simple execution engine which can execute the plans recommended by NoSE. One machine served to execute the statements and measure the execution time with another machine running an instance of Cassandra 2.0.9. Each machine has two six core Xeon E5-2620 processors operating at 2.10 GHz and 64 GB of memory. A 1TB SATA drive operating at 7,200 RPM was used for the Cassandra data directory. All Cassandra-level caching was disabled since we do not attempt to model the effects of caching in our cost model. A more complicated cost model which captures the effects of caching could be substituted into NoSE without changing the rest of the system.

We examined the RUBiS bidding workload by evaluating user transactions, which are groups of statements which would be executed for a single request to the application server. Results for the three schemas are shown in Figure 11. Mean response times for the different transaction types ranged from 1.25-113 ms for the schema recommended by NoSE, and 2.42-157 ms for the expert schema. We then measured the average response time for each transaction over 1,000 executions. The

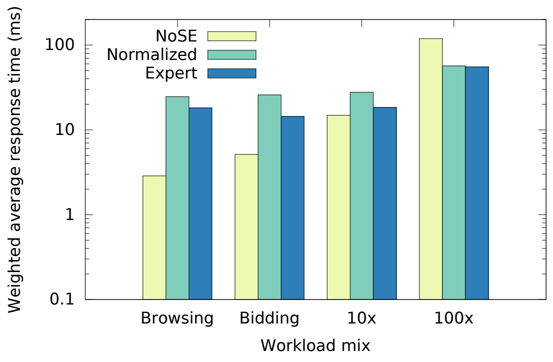


Fig. 12. Execution plan performance for varied write workloads

schema recommended by NoSE was able to achieve up to a $125\times$ improvement in performance over the expert schema for a single transaction. Additionally, NoSE was able to reduce the average response time weighted by the frequency of each transaction in the workload by a factor of 1.8. This is because NoSE was able to exploit information in the workload to perform more expensive updates for infrequent transactions (e.g., StoreBuyNow) or avoid denormalization for data which is frequently updated but infrequently read (e.g., AboutMe). By allowing more expensive updates which have little impact on the overall workload, the schema recommended by NoSE can contain additional column families to support frequently executed queries.

Furthermore, we examined the adaptability of NoSE to different workload distributions. We increased the weight of each RUBiS interaction involving writes by a factor of 10 and 100. In addition, we also tested NoSE against the RUBiS browsing workload mix which contains no updates. Each of these workload mixes leads to a different NoSE schema since writes becomes more or less expensive which changes the optimal level of denormalization. Results are shown in Figure 12. We note that in the $100\times$ case, NoSE performs worse than the expert schema. 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. Additionally, NoSE is not currently

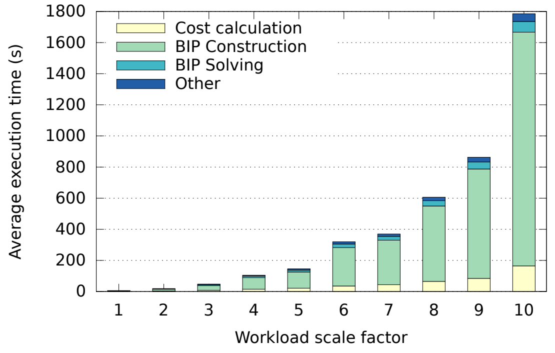


Fig. 13. Advisor runtime for varying workload sizes

capable of exploiting queries which make use of `GROUP BY` clauses. By using this knowledge, the expert schema is able to reduce the number of updates. For example, it is necessary to keep a record of which items a user has placed a bid on, but is not necessary for the application to efficiently retrieve all the individual bids for a given user. Since NoSE is unaware of the grouping of query results, we are unable to avoid storing data for individual bids, increasing the cost of writes.

B. Advisor Performance

Running NoSE for the RUBiS workload takes less than ten seconds. To evaluate the advisor runtime workloads larger than RUBiS, we generated random entity graphs and queries to use as input to our tool. The entity graph generation is based on the Watts-Strogatz random graph model [22]. 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. Statements are defined using a random walk through the graph to identify the path of the statement. For any statements involving a `WHERE` clause, we randomly generate three predicates along the statement path. Queries and updates have randomly chosen attributes along the path which are selected/updated.

Figure 13 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 figure shows the time required for NoSE to recommend a schema and a set of execution plans as a function of this factor. All experiments were run using a machine with the same specifications as in the previous section. Note that the runtime increases exponentially with the workload size, but this is independent of the weights of each of statement. The increase in runtime is a result of the increased number of support queries required to update the column families for each query. This interaction increases non-linearly with the workload size (e.g., increased numbers of queries and updates) since there are exponentially more ways that column families recommended for queries interact with updates. We note that the runtime of the BIP is relatively short, and the largest component runtime is the construction of the BIP and the associated constraints. There is likely to be room for optimization in the NoSE code to significantly reduce the runtime.

VIII. Related Work

Numerous tools are available, or have been proposed, 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 [8]–[15]. However, as was noted in Section II, 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 [23] or to determine a physical representation for the relations [24], [25]. There are also tools for automating relational partitioning and layout across servers [10], [26] or storage devices [27], [28]. DBDesigner [29] determines the physical representations (called projections) of tables for the Vertica [24] column, 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. A few relational design tools, including CoPhy [13], [15], CORADD [14], and a physical design technique for C-Store [25] have formulated the task of choosing a good set of candidates as a BIP. As was noted in Section V, Papadomanolakis and Ailamaki [13] presented a simple formulation of the problem as a binary integer program. CoPhy [15], an extension of this work, adds further optimizations to reduce calls to the relational query optimizer. As in our work, their approach exploits the decomposition of queries into components which can be analyzed independently. CoPhy also includes a rich set of constraints which may also be useful as extensions to NoSE.

Our approach to the schema design problem for extensible record stores owes an intellectual debt to GMAP [30], which was proposed as a technique for improving the physical data independence of relational database systems. In GMAP, both application queries and physical structures are described using a conceptual entity-relationship model. Queries are mapped to one or more physical structures which can be used to produce answers to the query. This approach is used 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 should be used to answer each query.

Others have also proposed writing queries directly against a conceptual schema. For example, ERQL [31], 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 all predicates given in a query must lie along the same path and 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 [32].

Vajk et al. [33] discuss schema design in a setting similar to ours. Their approach, like ours, starts with a conceptual schema. Queries are expressed in the UML Object Constraint Language. They sketch an algorithm that appears to involve the use of foreign key constraints in the conceptual schema 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 have also been proposed for adapting relational [34] and OLAP [35] schemas for NoSQL databases. However, these approaches are workload agnostic and do not necessarily produce schemas which can efficiently implement any particular workload. A workload-aware approach such as the one we take with NoSE is suggested as future work by Li [34].

IX. Conclusion

Schema design for NoSQL databases is a complex problem with many 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 various tradeoffs 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 be applied to other extensible record stores, such as HBase. We also intend to explore the use of a similar model on data stores with significantly different data models, such as key-value stores and document stores. Applying our approach to very similar data stores may only require changing the cost model and the physical representation of column families. However, we imagine more significant changes may be required to fully exploit the capabilities of different data models.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

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 http://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] E. Hewitt, *Cassandra: The Definitive Guide*, 2nd ed., M. Loukides, Ed. Sebastopol, CA: O’Reilly Media, 2011.

[8] 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.

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

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

[11] Benoit Dageville, D. Das, K. Dias, K. Yagoub, and M. Zait, “Automatic SQL tuning in Oracle 10g,” *PVLDB*, vol. 30, pp. 1098–1109, 2004.

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

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

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

[15] H. P. Company and A. Ailamaki, “CoPhy: A scalable, portable, and interactive index advisor for large workloads,” vol. 4, no. 6, pp. 362–372, 2011.

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

[17] DataStax, “About indexes in Cassandra,” 2014, retrieved Nov. 4, 2014 from http://www.datastax.com/docs/1.1/ddl/indexes.

[18] Gurobi Optimization, Inc., “Gurobi optimizer reference manual,” 2014, retrieved Aug. 8, 2014 from http://www.gurobi.com.

[19] M. Mior, “Nose - automated schema design for nosql applications,” 2016, retrieved Jan. 15, 2016 from https://github.com/michaelmior/NoSE.

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

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

[22] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’ networks,” *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

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

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

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

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

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

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

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

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

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

[32] 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.

[33] 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.

[34] C. Li, “Transforming relational database into hbase: A case study,” in *Proc. ICSESS 2010*, 2010, pp. 683–687.

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

APPENDIX

A. Support Query Generation

A support query to perform the update on the materialized view for the example shown in Figure 9 is given below:

```
SELECT Room.RoomID, Room.RoomFloor,
PointsOfInterest.POIID, Hotel.HotelID,
Room.RoomID
FROM PointsOfInterest.Hotels
.Rooms.Reservations.Guests
WHERE Guest.GuestID=?id AND
Room.RoomRate=?rate2
```

This support query simply extends the path given in the update to incorporate the attributes being selected and uses the same set of predicates. All the necessary attributes (the primary key of the materialized view) are selected. After performing this query, the entries corresponding to the old values of `RoomRate` would be removed. These entries which were selected are then modified in-place and reinserted into the column family. Note that while it is possible to perform these operations atomically, our approach currently does not provide any isolation. Below we provide details on the support queries required for different statements.

In all cases below, a support query does not need to fetch attributes which are given in the `WHERE` clause of a statement. If all the necessary attributes are given in a `WHERE` clause, then no support queries are required. Note that since our query language does not allow branching, it may be necessary to perform multiple support queries to select attributes from entities along two different paths. If a statement requires multiple support queries, we take the cross product of the results to account of multiple paths between entities on either side of the branch point.

1) `INSERT` and `CONNECT`: To construct a support query for a `CONNECT` statement, we first split the path for the column family at the relationship where the connection is being added. Two support queries are then constructed, one on each side of the path split. Each of these support queries selects attributes for entities along the path using a predicate on the primary keys specified in the `CONNECT` statement. `INSERT` statements only require support queries if they come with an accompanying `CONNECT` clause. In this case, the support queries function identically with the only difference being that it is not necessary to select attributes for the entity being inserted since they are provided in the `INSERT` statement.

2) `UPDATE`: The support queries required for updates depends the attributes what attributes are updated. If an update modifies key attributes of the column family, we need to select all attributes since it is necessary to perform a deletion followed by an insertion of a new record to modify the keys. If an update only modifies non-key attributes, support queries only need to select the key attributes. In this case, we can perform the update directly against the backend database.

3) `DELETE` and `DISCONNECT`: Support queries for `DELETE` and `DISCONNECT` statements only need to select the key attributes for the column family being modified. In both cases, we perform a delete against the backend database and only require the keys to perform this deletion.

B. Cost Model

The BIP constants C_{ij} are used to represent the cost of using a particular column family in the plan for a particular query. NoSE derives these constants as a byproduct of plan enumeration. Whenever a `get` request is made for a column family, we consider the cost as a function of the number of entries retrieved in the column family (partition key, clustering key, and value).

1) Calibration: The `get` operations executed by our system can be characterized by a number of simple parameters. We have constructed a set of simple experiments which both validate the cost model as well as provide values for the parameters in a given configuration. For this evaluation, we assume that target applications deal with relatively small amounts of data per query result (both in size and number of attributes). All Cassandra queries under our execution model specify the partition key and some prefix of the clustering keys. We assume that the cost of a query is a fixed cost for each partition key, plus a variable cost dependent on the number of rows retrieved by each query. We can estimate the rows retrieved by each query based on the cardinality of each attribute while incorporating the filtering which takes place on the clustering keys.

To validate this model, we constructed a synthetic database consisting of a single column family with a number of partitions and one column in the clustering key. We can control the number of results for a query for an individual partition key by varying the cardinality the clustering key (there is a single result for each combination of partition key and clustering key). We then calculate the response time of queries over several random keys with a different number of rows which can be processed while varying the size of the result set. Fitting a line to these results gives the parameters necessary for our model. Our model has three parameters. The first is a fixed cost for each column family used to answer a particular query. The other parameters are variable costs for each partition key and row which are accessed by each lookup.

We leave an examination of the sensitivity of this model and its parameters to a particular configuration of Cassandra as future work. However, we note that NoSE is not dependent on any properties of this particular cost model. NoSE would function equally well with any other cost model which may capture different the effect of Cassandra configurations.

2) Query Plan Cost Composition: We calculate the cost of performing all the `get` requests for a query by estimating the cardinality of the result set. For this we use the metadata associated with the column family. The cardinality of attributes is used to estimate the percentage of entities filtered by equality predicates with a constant estimate of 10% used for range predicates. We then multiply this by the estimated number of entities in the target entity set which is also given as metadata in the conceptual model.

This estimation is performed for each step through the entity graph along the query path. When traversing foreign key relationships, we estimate the cardinality of the first entity set. We then calculate the expected number of entities which will be retrieved from the second entity set based on sampling with replacement.

Plans for relaxed prefix queries which require filtering by the application are accounted for by adjusting the estimated cardinality to exclude the attribute used for filtering. That is, we assume that discarding tuples which are filtered is "free", with the cost of this step accounted for by the additional data fetched from the database which is subsequently discarded.

The cost of a relaxed prefix query involving a sort is simply a small constant to express that sorting is more expensive than retrieving sorted data directly from the database. A more accurate cost model would be necessarily be environment-specific to capture the tradeoff between network bandwidth and the execution time of sorting.

With these estimated costs tracked during query plan space enumeration, the constants C_{ij} is extracted simply by summing the cost of all `get` operations for query i on column family j. This is possible since each column family will only appear once in the tree of enumerated query implementation plans. Furthermore, the cost of executing a prefix query is independent of the cost of executing the remainder query so the cost C_{ij} is independent of $C_{ij'}$ for all $j \neq j'$.

C. Schema and Implementation Descriptions

Below are a description of the three schemas and query implementation plans used in our evaluation. The underlined attributes of column families form the primary key. Italicized attributes are part of the clustering key with non-italicized attributes forming the partition key. The path traversed by each column family is implicitly given in the column family name so it is omitted from this description.

1) Recommended by NoSE:

Column Families

categories
categories.dummy, *categories.id*, categories.name

bids_by_date
items.id, *bids.date*, *bids.id*, *users.id*, users.nickname, bids.qty, bids.bid

items_by_id
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

categorized_items
categories.id, *items.id*, items.name, items.initial_price, items.max_bid, items.nb_of_bids, items.end_date

users_by_id
users.id, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date

item_bids
items.id, *bids.bid*, *bids.id*, bids.qty, bids.date

comments_to_user
users.id, *comments.id*, comments.rating, comments.date, comments.comment

commenting_user
comments.id, *users.id*, users.nickname

user_bought_now
users.id, *buynow.date*, *buynow.id*, buynow.qty

bought_now_item
buynow.id, *buynow.date*, *items.id*

user_items_sold
users.id, *items.end_date*, *items.id*

user_bids
users.id, *items.end_date*, *bids.id*, *items.id*

item_category
items.id, *categories.id*

item_seller
items.id, *users.id*, items.end_date

bids_with_user
items.id, *bids.id*, *users.id*, items.end_date

Plans

BrowseCategories
Get from **users_by_id**
Get from **categories**

ViewBidHistory
Get from **items_by_id**
Get from **bids_by_date**

ViewItem
Get from **items_by_id**
Get from **bids_by_date**

SearchItemsByCategory
Get from **categorized_items**, Filter by items.end_date

ViewUserInfo
Get from **users_by_id**
Get from **comments_to_user**

BuyNow
Get from **users_by_id**
Get from **items_by_id**

StoreBuyNow
Get from **items_by_id**
Insert into **items_by_id**
Get from **item_category**, Insert into **categorized_items**
Get from **item_seller**, Delete from **user_items_sold**, Insert into **user_items_sold**
Get from **bids_with_user**, Delete from **user_bids**, Insert into **user_bids**
Get from **item_seller**, Insert into **item_seller**
Get from **bids_with_user**, Insert into **bids_with_user**
Insert into **user_bought_now**
Insert into **bought_now_item**

PutBid
Get from **users_by_id**
Get from **items_by_id**
Get from **item_bids**

StoreBid
Get from **items_by_id**
Get from **users_by_id**, Insert into **bids_by_date**

Insert into **item_bids**
Get from **item_seller**, Insert into **user_bids**
Get from **item_seller**, Insert into **bids_with_user**
Insert into **items_by_id**
Get from **item_category**, Insert into **categorized_items**

PutComment
Get from **users_by_id**
Get from **items_by_id**
Get from **users_by_id**

StoreComment
Get from **users_by_id**
Insert into **users_by_id**
Insert into **comments_to_user**
Get from **users_by_id**, Insert into **commenting_user**

AboutMe
Get from **users_by_id**
Get from **comments_to_user**
Get from **commenting_user**
Get from **user_bought_now**, Get from **bought_now_item**, Get from **items_by_id**
Get from **user_items_sold**, Get from **items_by_id**
Get from **user_bids**, Get from **items_by_id**

RegisterItem
Insert into **items_by_id**
Insert into **categorized_items**
Insert into **user_items_sold**
Insert into **item_category**
Insert into **item_seller**

RegisterUser
Insert into **users_by_id**

2) Normalized:

Column Families

user_data
users.id, *regions.id*, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date, regions.name

user_buynow
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty

user_items_bid_on
users.id, *items.end_date*, *bids.id*, *items.id*, bids.qty

user_items_sold
users.id, *items.end_date*, *items.id*

user_comments_received
users.id, *comments.id*, *items.id*, comments.rating, comments.date, comments.comment

commenter
comments.id, *users.id*, users.nickname

items_with_category
items.id, *categories.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

items_data
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

item_bids
items.id, *bids.id*, *users.id*, items.max_bid, users.nickname, bids.qty, bids.bid, bids.date

items_by_category
categories.id, *items.end_date*, *items.id*

category_list
categories.dummy, *categories.id*, categories.name

regions
regions.id, regions.name

Plans

BrowseCategories
Get from **user_data**
Get from **category_list**

ViewBidHistory
Get from **items_with_category**
Get from **item_bids**

ViewItem
Get from **items_with_category**
Get from **item_bids**

SearchItemsByCategory
Get from **items_by_category**, Get from **items_with_category**

ViewUserInfo
Get from **user_data**
Get from **user_comments_received**, Get from **commenter**

BuyNow
Get from **user_data**
Get from **items_with_category**

PutBid
Get from **user_data**
Get from **items_with_category**
Get from **item_bids**

PutComment
Get from **user_data**
Get from **items_with_category**
Get from **user_data**

AboutMe
Get from **user_data**
Get from **user_comments_received**, Get from **commenter**
Get from **user_buynow**, Get from **items_with_category**
Get from **user_items_sold**, Get from **items_with_category**
Get from **user_items_bid_on**, Get from **items_with_category**

RegisterItem
Insert into **items_with_category**
Insert into **user_items_sold**
Insert into **items_by_category**

RegisterUser
Get from **regions**, Insert into **user_data**

StoreBuyNow
Get from **items_with_category**, Insert into **items_with_category**, Delete from **items_by_category**, Insert into **items_by_category**
Insert into **user_buynow**

StoreBid
Get from **item_bids**, Insert into **item_bids**
Get from **items_with_category**, Insert into **items_with_category**
Insert into **user_items_bid_on**

StoreComment
Get from **user_data**, Insert into **user_data**
Insert into **user_comments_received**

3) Expert:

Column Families

user_data
users.id, *regions.id*, users.firstname, users.lastname, users.nickname, users.password, users.email, users.rating, users.balance, users.creation_date, regions.name

user_buynow
users.id, *buynow.date*, *buynow.id*, *items.id*, buynow.qty

user_items_bid_on
users.id, *items.end_date*, *bids.id*, *items.id*, bids.qty

user_items_sold
users.id, *items.end_date*, *items.id*

user_comments_received
users.id, *comments.id*, *items.id*, comments.rating, comments.date, comments.comment

commenter
comments.id, *users.id*, users.nickname

items_with_category
items.id, *categories.id*, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

items_data
items.id, items.name, items.description, items.initial_price, items.quantity, items.reserve_price, items.buy_now, items.nb_of_bids, items.max_bid, items.start_date, items.end_date

item_bids
items.id, *bids.id*, *users.id*, items.max_bid, users.nickname, bids.qty, bids.bid, bids.date

items_by_category
categories.id, *items.end_date*, *items.id*

category_list
categories.dummy, *categories.id*, categories.name

regions
regions.id, regions.name

Plans

BrowseCategories
Get from **user_data**
Get from **category_list**

ViewBidHistory
Get from **items_with_category**
Get from **item_bids**

ViewItem
Get from **items_with_category**
Get from **item_bids**

SearchItemsByCategory
Get from **items_by_category**, Get from **items_with_category**

ViewUserInfo
Get from **user_data**
Get from **user_comments_received**, Get from **commenter**

BuyNow
Get from **user_data**
Get from **items_with_category**

PutBid
Get from **user_data**
Get from **items_with_category**
Get from **item_bids**

PutComment
Get from **user_data**
Get from **items_with_category**
Get from **user_data**

AboutMe
Get from **user_data**
Get from **user_comments_received**, Get from **commenter**
Get from **user_buynow**, Get from **items_with_category**
Get from **user_items_sold**, Get from **items_with_category**
Get from **user_items_bid_on**, Get from **items_with_category**

RegisterItem
Insert into **items_with_category**
Insert into **user_items_sold**
Insert into **items_by_category**

RegisterUser
Get from **regions**, Insert into **user_data**

StoreBuyNow
Get from **items_with_category**, Insert into **items_with_category**, Delete from **items_by_category**, Insert into **items_by_category**
Insert into **user_buynow**

StoreBid
Get from **item_bids**, Insert into **item_bids**
Get from **items_with_category**, Insert into **items_with_category**
Insert into **user_items_bid_on**

StoreComment
Get from **user_data**, Insert into **user_data**
Insert into **user_comments_received**