A Logical Design Process for Columnar Databases

João Paulo Poffo* and Ronaldo dos Santos Mello†
Informatics and Statistics Department
Federal University of Santa Catarina
Florianópolis/SC, Brazil 88040-90
Email: jopapo@gmail.com* and r.mello@ufsc.br†

***Abstract*—Emerging technologies often break paradigms. NoSQL is one of them and is gaining space with the raising of Big Data, where the data volume exceeded the petabyte frontier and the information within these data can be of great importance to strategic decisions. In this case, legacy relational databases show themselves inadequate to efficiently manage these data and, consequently, their traditional project methodologies should must be reviewed to be suitable to new data models, such as the NoSQL columnar model. Regarding columnar database design, the literature lacks of methodologies for logical design, i.e., processes that convert a conceptual schema to a logical schema that optimize access and storage. Thus, this work proposes an approach for logical design of columnar databases that contributes to fill the void between classic project methodologies and the technological forefront with the NoSQL movement, in particular, columnar databases. Preliminary experiments had demonstrated that the methodology is promising, if compared with a baseline.**

***Keywords*–database design; logical design; nosql; columnar database.**

I. Introduction

With the advent of the cloud computing paradigm, the opportunity to provide DataBase Management Systems (DBMS) as services is strengthened, as witnessed by Amazon RDS and Microsoft SQL Azure [1]. NoSQL is one of these movements which is standing out by providing DBs with high availability and scalability. These characteristics are essential to social media, profile repositories, content providers, among other applications [2].

NoSQL is a commonly known term that covers several non relational DBs which can manage high amounts of data. They are categorized by key/value, column, document and graph DBs [3][4]. In *DB-Engines* [5], there is a ranking of DB products and in the top ten, seven of them are relational. However, what draws the attention are the three others: MongoDB (a document DB), Cassandra (a columnar DB) and Redis (a key/value DB).

A DB is called columnar when the smallest information unit to be manipulated is a column. The best way to imagine it is like a two-level data aggregation structure [6]. As in key/value DBs, the first level is a key that identifies an aggregation of interest. The difference with respect to columnar DBs is that the second level contains several columns that can hold simple or complex values, and these columns can be accessed all at once or one at a time.

The traditional DB design methodology has three main phases: conceptual, logical and physical design [7]. In contrast, this sequence seems to have been suppressed for columnar DBs. It neglects the conceptual design phase, starting with the column sets design and how they will be accessed [8].

Based on this motivation, this work proposes a reconciliation between the classical DB design approach and columnar DBs, contributing with a logical design process that considers the semantics of the application domain (a conceptual schema) and aims to achieve an optimized conversion from a conceptual schema to a logical columnar schema. A conceptual model must be expressive, simple, minimal and formal [9] and there are several models that respect these standards. In this work, the Extended Entity-Relationship (EER) conceptual model is considered. We also propose several conversion rules from an EER conceptual schema to a logical notation suitable to the columnar data model. This logical data model is another contribution of this work, which can also be applied to represent a reverse engineered schema of a columnar DB. The most adherent usage of our approach is in long-term running high-growth applications that needs to scale, like never-ending games and social network, among others. All these features are demonstrated through an experimental evaluation.

Preliminary related work are [10], where the conceptual model are mapped into hierarchical model (XML), and [11], who proposes to do the same, but targeting object-oriented DBs. However, they do not focus on NoSQL DB design. We just borrow from them some ideas for the proposed conversion rules. In [12], it is presented a *NoSQL Abstract Model (NoAM)* which aims to embrace the data model of any existing NoSQL DB, and [13] focuses on Cassandra columnar DBMS. Sharp et al. [14] and Schram et al. [15] suggest limited orientations for the logical and physical design with columnar DBs. However, all of them lack information about how to provide logical design based on a conceptual schema. Other works [4][8][15] deal with logical design using columnar DBs, but do not present detailed conversion rules as well as an evaluation of their proposals. Distinctly, Meijer and Bierman [16] present a mathematical model to NoSQL DBs and demonstrates their correlation with the relational model. However, it does not make reference to columnar DBs nor deals with conceptual design or conversion process. In short, the literature still lacks a comprehensive approach to this problem.

The rest of this paper is organized as follows. Next section analyzes related work, exposing their strengths and weakness, as well as the gap filled with this work. In Section III, fundamentals about DB design and NoSQL, with emphasis in columnar DBs, are presented. Section IV is dedicated to our proposal for logical design of columnar DBs, including its formalization. Some experiments are designed and evaluated in Section V, followed by our conclusions in Section VI.

II. Related Work

Besides the classical methodology for relational DB design [7], some conceptual to logical conversion for non-relational DBs proposals are found, such as XML DBs [10], object-oriented DBs [11] and NoSQL DBs [12][13]. Still, there are several guidelines for how to directly convert some

TABLE I. RELATED WORK COMPARISON.

Feature / **Work**	Schroeder & Mello 2008 [10]	Fong 1995 [11]	Sharp et al. 2013 [14]	Schram & Anderson 2012 [15]	Wang & Tang 2012 [8]	Kaur & Rani 2013 [4]	Meijer & Bierman 2011 [16]	Bugiotti et al. 2014 [12]	Chebotko, Kashlev & Lu 2015 [13]	This work 2016
Conceptual design	●	●	○	○	◐	◐	◐	●	◐	●
Logical design	●	●	●	●	◐	◐	◐	●	●	●
Conversion rules	●	●	○	○	○	◐	○	●	●	●
Columnar DB	○	○	●	●	●	○	○	◐	●	●
Validation	●	●	○	○	○	◐	●	●	◐	●

logical structures in columnar DBs [4][8][14][15][17]. What is evident in this current literature is the lack of a clear and comprehensive approach that transforms a conceptual schema, such as an EER schema, into a logical schema for columnar DBs.

Schroeder and Mello [10] proposes a mapping approach from a conceptual model (EER) to an equivalent XML logical model. Its process comprises conversion rules for all EER concepts and it is improved by considering workload information. Despite its different focus, its methodology is well-suited to convert complex objects as our work. The same applies to [11], whose target is an object-oriented DBs. Both have a comprehensive EER-to-logical conversion approach into their specific outputs, but they do not explore NoSQL DBs.

The proposal [12] stands for a NoSQL DB design solution to any kind of NoSQL data model. The basis of their approach is what they call an abstract data model using aggregates called NoAM. Their process considers a conceptual data model, a design of aggregated objects in NoAM, a high level NoSQL DB design and its implementation. Although the proposal acts in the three design phases, it does not focus on columnar DBs, and does not consider all the conceptual constructs, such as composite attributes and N:M relationships, nor formalizes conversion rules between a conceptual modeling and logical representations in the NoAM model. The approach in [13] is similar but more complete, as it covers all concepts from an ER conceptual model (without extensions introduced by EER) for a logical design. A logical columnar DB schema for Cassandra is also proposed. The proposed conversion is query-oriented and enforces redundancy, which is consistent with three followed assumptions: *know your data, know your queries, aggregate and duplicate your data*. Our approach differs from this one by not considering aggregates and being validated experimentally.

Taking a look now at industry efforts, Microsoft presents detailed instructions to the creation of columnar DBs optimized for writing and reading [14]. These guidelines are enriched by considering the *Wide-Column* concept, which is similar to a matrix transposition, i.e., the focus is on the columns instead of the rows. They determine what must be done to maximize scalability, availability and consistency, but they lack the conceptual data modeling related to the domain.

The case study in [15] presents a system whose relational access and data volume grows very fast (*Twitter* data). So, the authors propose the usage of a columnar DBMS (Cassandra). All of the study, challenges and system construction are discussed. They explain how to perform the transition to the columnar DB and their results. They manually use workload information to optimize the design by applying denormalization. They also presents a practical case and its challenges, but they lack a formalization and validation of the process. Besides, they also suppress the conceptual design phase.

Similarly, only to contrast relational and non-relational approaches, Wang and Tang [8] show simple principles of conceptual design using UML and the straightforward conversion to Cassandra. However, their proposal considers a very restrict set of conceptual structures, and does not formalizes its process. In [4], the conceptual design (based on the ER model) of a case study for an application related to *blog posts* is converted to MongoDB (a document DB) and Neo4j (a graph DB). Their focus is not on the conversion itself, but the access optimization, enriching our process with their troubleshooting. Differently, the approach in [16], named *CoSQL*, presents a mathematical model to key/value DBs and demonstrates its correlation with the relational model. It also defines a common query language to relational and non-relational DBs based on the relational algebra. Its proposal to defined a logical layer had inspired our conversion process by proving a conceptual-to-logical correlation for NoSQL DBs. However, this approach does not formalize a conversion process nor validate it.

Table I shows a comparison of related work. It highlights five features: *Conceptual design* indicates that there is at least one kind of conceptual model considered by the work; *Logical design* indicates that the related work considers logical design; *Conversion rules* indicate if the work clearly defines rules to transform a conceptual schema into a logical schema; *Columnar DB* focuses on this kind of DB and, lastly, if the approach reports some kind of *Validation*. To each feature is assigned one of three signs: the work fully supports the feature ●, the feature is not mentioned ○ or the feature is partially treated ◐.

As shown in Table I, only our proposal fully covers all the considered features. The formalization of the conceptual-to-logical conversion rules, the definition of the conversion process and its validation are the main contributions of this work. These points are detailed in Section IV.

III. Fundamentals

This section presents the fundamentals related to the classical DB design, NoSQL DBs and columnar DBs.

A. *DB Design*

The DB design aims to rightly define real world facts and their relationships, as well as their modeling in a target DB, aiming at maximizing storage and access requirements. The classical phases of DB design are: data requirements gathering, conceptual, logical and physical design [7].

Several conceptual models are available for DB design, like Unified Modeling Language (UML), Object with Roles Model (ORM), and the most representative one, the EER [7][9]. The three main EER concepts are: *(i)* entity (an abstraction of a set of similar real world objects), *(ii)* relationship (a semantic connection between entities), and *(iii)* attribute (a property associated with an entity or relationship).

Figure 1 shows an example of an EER schema. In this example we can identify abstract constructs such as classification (entities and their attributes), aggregation (composite attributes or association relationships) and generalization (subset and superset behavior) with associated constraints. In traditional

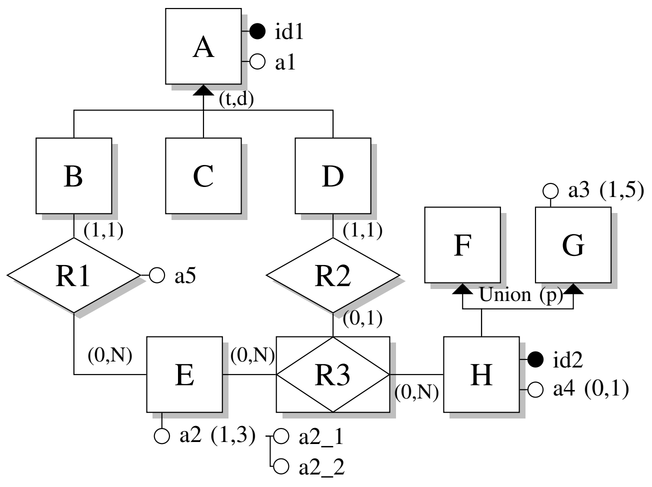


Figure 1. An example of EER schema [10].

DB design methodologies, this conceptual schema is the basis for generating a logical and then a physical schema in the target DB model. In the first case, this mapping is supported by a conversion process that offer alternatives to generate an optimized DB schema.

B. *NoSQL*

A DB is based on a data model and a set of operations that allow data definition and manipulation. Data manipulation, in particular, respects the classical Atomicity, Consistency, Isolation and Durability (ACID) properties [6]. Until recently, this fundamental and untouchable acronym ruled sovereign.

However, digital data show today a fast growth in Volume, Variety and Velocity (VVV). Such a phenomena is called *Big Data*, which typically corresponds to massive data collections that cannot be suitable handled by traditional DBMSs that respect to the ACID properties. In order to address Big Data management, movements like *NoSQL* and *NewSQL* had emerged [3].

NoSQL, in particular, covers a wide range of technologies, architectures and data models. NoSQL DBs usually do not ensure ACID properties in order to avoid the overhead to guarantee them and provide better scalability and availability [18]. Instead, they are Basically Available, hold a Soft state and are Eventually consistent (BASE), i.e., availability and partitioning are prioritized to the detriment of consistency.

NoSQL DBs comprises the following data models [3]: *(i)* **key/value** DBs store data items identified by a key and indexed by hash tables. Values can contain both simple and complex data, but are accessed as a single unit. Queries are usually only directly by key; *(ii)* **columnar** DBs store heterogeneous sets of columns for each data item. Each column holds a simple value or, in some cases, a set of nested columns; *(iii)* **document** DBs store data items that are called documents, which are usually stored in XML, JSON or BSON formats. Unlike key/value DB, values can be semistructured and one document usually hold a set of attributes; *(iv)* **graph** DBs maintain nodes and edges, and both can hold attributes. It is the only one that support explicit relationships.

The focus of this paper is on columnar DBs, which is detailed in the following.

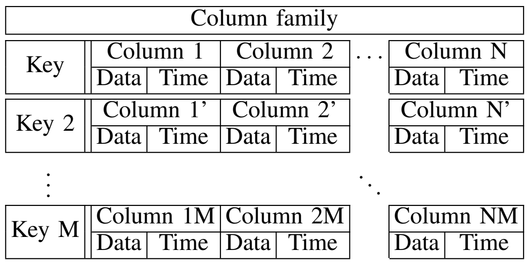


Figure 2. Columnar DB data model representation.

C. *Columnar DBs*

Columnar DBs (also known as extensible records, wide-column, column-family, column-oriented or BigTable-implementations) are so named because the smallest portion of information increment is a column. Each column has basically a name and a value, and a column value can hold a simple value or a set of columns, as stated before. A column also contains a timestamp which is used to manage mutual exclusion regarding concurrency problems.

A column family is a container of lexical ordered columns [19]. Thus, columns that are read together must be kept in the same column family. It is possible to add undeclared columns to a column family. Its flexible structure allows it. So, it is frequently sparse. A key uniquely identifies each line in a column family. This key can define, according to its partitioning strategy, in which cluster server the data are stored. The same key can be used in different column families. Figure 2 shows a basic representation of this data model.

Cassandra is a popular columnar DBs, originally created by Facebook and now maintained by Apache [20]. It has some special features like super column and composite keys. The former works like a nested column family. The latter is a way to add one dimension to the key into a column family.

It is important to empathize that there is not a global rule or standard with respect to the columnar DB data model. This is highlighted by Table II, that shows which concepts are supported by the main columnar DBMS. It indicates three kinds of information regarding each DB product: it supports the concept ●, it does not support it ○, or it's possible to workaround the concept or it exists in a limited way ◐.

TABLE II. DATA MODEL CONCEPTS FOR COLUMNAR DATABASES.

Feature / **DB**	Cassandra	Riak	HBase	DynamoDB	Accumulo	Teradata	SybaseIQ
Collection data type	●	●	●	●	◐	●	●
Flexible structure	●	●	●	●	●	●	○
Composite keys	●	○	○	●	○	●	●
Super columns	●	○	○	○	○	○	○

Columnar DBs can be horizontally and vertically partitioned. Some of the good candidates to use this kind of DB are logs, content providers, personal pages, blogs, among others [6][21]. On the other hand, columnar DBs are not a good choice when the scope of a system is not clear, because of the high cost on deep structural changes. Despite flexible, changes almost always must be adjusted in the application and may deteriorate its performance. This is not the case of classical relational approach because of its rigid structure. Therefore, columnar DBs are more sensitive to query pattern changes than

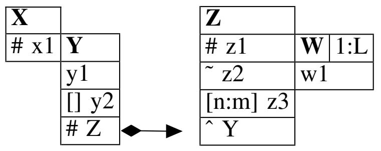


Figure 3. Overview of the diagrammatic notation of a columnar DB schema.

in the schema, other than relational DBs. This fact strengthens the need for a well-defined conceptual and logical design.

IV. Proposal

This section details our approach for logical design of columnar DBs. First, it introduces a logical design notation for columnar DBs. In the following, the overall conversion process of an EER conceptual schema to a logical columnar DB schema is defined in terms of high level algorithms, inspired by [9].

The general reasoning of our conversion process is to offer a logical schema with data sets that support nested column families and bidirectional relations. The first strategy is achieved by the use of *shared keys*, which provides access efficiency as close related data are stored near each other. The second strategy is used when the first strategy is not applied, being achieved through *artificial relations*. Both concepts are detailed in the following. We decided not to consider the traditional aggregate approach for logical modeling of NoSQL databases, followed by most of the related work, because it can generate deep nested data relationships which cannot be efficient, in terms of accessing, for some application domains, as illustrated in our experimental evaluation (see Section V).

A. Logical Notation for Columnar DBs

The conceptual modeling represents relevant data but not how they are persisted in the DB [7]. So, it is necessary to provide some abstraction level of the DB to the user. The generation of this abstraction level is called logical design, and it comprises the transformation of a conceptual schema into a logical schema suitable to the DB data representation.

The conceptual-to-logical conversion is a transformation between data models in different abstraction levels. As there is no a standard for the columnar DB data model, we define a logical notation for this data model. The concepts of our logical notation are defined in the following.

Definition 1 (Column): A column `c` is a tuple $c = \{(n, v, t) | n = \text{name}, v = \text{value}, t = \text{timestamp}\}$.

Definition 2 (Column family): A column family is a map $f : A \mapsto B$ where `A` = {key} is a set of unique keys, and `B` = {c} is a set of columns such that, for every $\alpha \in$ `A`, there is a unique object $f(\alpha) \in$ `B`.

Definition 3 (Shared key): Given a logical columnar DB schema `S` so that $F \in S$ is a column family and $F' \in S$ is another column family, a key is shared if $key(F) = key(F')$, i.e., a key is a shared key if the same value is used the identifier of two or more column families.

Definition 4 (Artificial relation): Given a logical columnar DB schema `S` so that $F \in S$ is a column family and $F' \in S$ is another column family, there is an artificial relation if $c_i \in F = key(F')$, i.e., if any column in a family match the key of a column family.

Figure 3 shows the notation for the logical representation of a columnar DB schema proposed in this paper. A column family is represented by a rectangle with the name on top and an optional cardinality constraint in its side. This cardinality constraint allows the definition of repeatable columns within a family when a column family is nested into another one (`W`, for example). Each internal line in a column family represents a column where the hash symbol (#) means the key, tilde (˜) means a mandatory column, caret (ˆ) means an artificial relation and brackets ([]) define unrestricted internal collections. Brackets with values ([n:m]) mean an explicit cardinality constraint. No column data types are described in this version of the logical notation as NoSQL DBs supports virtually anything.

A shared key is the reuse of the key of another column family. It is represented by the coupling of two or more column families in a way that its hierarchy is visible. In other words, a column family that intends to share the key of another one is welded with the column family that holds the original key, like `X` to `Y`, and `Z` to `W`. The artificial relation is represented by a line that connects column families whose tips are arrows or diamonds, like `Z` to `Y`. The arrow means the existence of a column with the caret (that stores the key of the other family) and the diamond means an aggregation on the column. Finally, the 1:L relationship (also introduced in this work) represents a not known superior limit, like 1:N relationships, but this limit is not high. Usually the "L" side is associated to weak entities (employee dependents, for example). It allows the use of shared keys, which cannot be used in 1:N relationships.

B. Conversion Process

It is important to notice that, due to the flexible nature of columnar DBs, it does not enforce several restrictions. The application must be responsible for it. It is argued in [22] that the absence of a DB schema is a fallacy. A schema always exists, but instead of being enforced by the DB, the application assumes the control of data integrity constraints. In this context, the concept of shared key, as introduced before, was defined to deal with the absence of referential integrity in columnar DBs. This strategy avoids the number of physical references between column families, and, as a consequence, the overhead to manage the referential integrity.

The high level algorithms for mapping EER constructs to the columnar DB logical representation are presented in the next Sections. These algorithms are based on the notion of entity paternity, which is defined as follows.

Definition 5 (Entity Paternity): Given two entities E_P and E_C, we say that E_P is parent of E_C (or, in other words, E_C is child of E_P), if: (a) E_C is a specialized entity and E_P is the generic entity in a specialization relationship; (b) E_P is the entity that unifies two or more entities in an union relationship, being E_C one of the unified entities; (c) E_P is a mandatory entity that has 1 as maximum cardinality *on its side* in a `1:1`, `1:N` or N-ary relationship.

In short, our process traverses all entities and, for each entity, it checks if there is a relationship where this entity is a child. If it exists, the parent entity in the relationship is prioritized, i.e., it is converted first. This checking and prioritization is repeated until there is no more parent entities.

When all column families are generated from entities (in a recursive way that takes into account paternity relationships), column family keys are defined and entity attributes are converted. The process ends with the conversion of the relationships, which can generate new columns and column families to represent adequately its dependencies. All of this conversion reasoning is detailed in the following.

C. General Conceptual to Logical Schema Conversion

The algorithm in Figure 4 is the main conversion process. Its input is an EER schema. All the entities of the EER schema are traversed$^{:2\text{ (line 2)}}$ and, for each one, the algorithm in Figure 6 is triggered$^{:3}$. Its output is added to the set of column families that compose the columnar DB logical schema and then returned$^{:5}$.

```
  Input: EER schema ($\alpha$)
  Output: Columnar DB logical schema ($\alpha'$)
1 $\alpha' \leftarrow \emptyset$
2 foreach $\epsilon \in \alpha | \epsilon$ is an entity do
3     $\alpha' \leftarrow \alpha' \cup createFamily(\epsilon)$
4 end
5 return $\alpha'$
```

Figure 4. Conceptual to logical schema conversion

Example 1 (Schema conversion): The conversion of the EER schema of Figure 1 generates the columnar DB logical schema in Figure 5 according to algorithm in Figure 4.

When an entity is visited by the loop, its parent entity is converted before it, in a recursive way, if it exists (see algorithm in Figure 6). Each converted entity is marked to avoid its repeated conversion. The loop in this algorithm aims to reach all EER schema entities, even the ones that are part of disjoint groups of related entities. A general example is given in the following, and details about the conversion of each EER construct are further exemplified.

D. Column Family Generation

The input of algorithm in Figure 6 can be an entity or a relationship and it outputs a set of column families. For each analyzed entity, this algorithm provides the conversion of all other conceptual constructs related to it (relationship types and attributes). The same holds if a relationship is being treated. First, it initializes the set of output column families$^{:1}$. If the entity or relationship was not visited yet$^{:2}$, then a list of EER concepts is created with generalizations, unions and association relationships where the input entity is child$^{:3}$. If the input is a relationship, it is assumed that it does not have relationships, so the list is empty.

Then, for each concept$^{:4}$, the same algorithm is triggered recursively with its parent as input$^{:5}$. Next, a new column family is created$^{:7}$ with a name$^{:8}$ and a key (see algorithm in Figure 7). In the following, for each attribute of the input$^{:10}$, algorithm in Figure 9 is called$^{:11}$ to define a suitable column (or even an aggregated column family) to the new column family. The new column family is added to the result set$^{:13}$, followed by the traversing of the list created before, as well as reflexive or `N:M` relationships of the input entity$^{:14}$. Thus, for each relationship, algorithm in Figure 10 is triggered with the relationship and the new column family. Finally, its output is added to the result set$^{:15}$ and the result set is returned$^{:18}$.

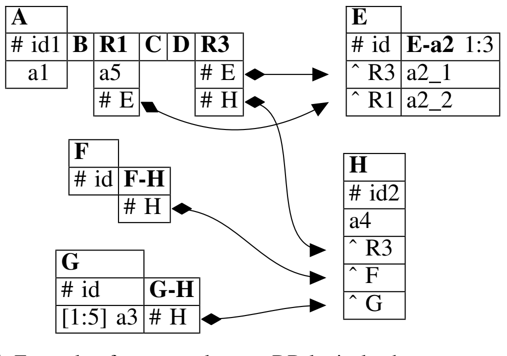


Figure 5. Example of output columnar DB logical schema generated from the EER schema of Figure 1.

```
   Input: Entity or relationship ($\epsilon$)
   Output: Column family set ($\omega$)
1  $\omega \leftarrow$ Preexistent column family referring to $\epsilon$ or empty set
2  if checkAndMarkIfIsFirstVisitTo($\epsilon$) then
3      $\pi_P \leftarrow$ generalizations, unions, n-ary and binary parent
         relationships of $\epsilon$, in this order.
4      foreach $\pi \in \pi_P$ do
5          $\omega \leftarrow \omega \cup createFamily($Parent entity in $\pi)$
6      end
7      $\epsilon' \leftarrow$ New empty column family
8      $\epsilon'.Name \leftarrow \epsilon.Name$
9      $\epsilon'.Key \leftarrow defineKey(\epsilon)$
10     foreach $\delta \in \epsilon | \delta$ is an attribute do
11         $\omega \leftarrow \omega \cup convertAttribute(\epsilon', \delta)$
12     end
13     $\omega \leftarrow \omega \cup \epsilon'$
14     foreach $\pi \in \epsilon | \pi \in \pi_P \vee \pi.Type \in \{reflexive, N : M\}$ do
15         $\omega \leftarrow \omega \cup convertRelationship(\pi, \epsilon')$
16     end
17 end
18 return $\omega$
```

Figure 6. Algorithm for entity or relationship to column family conversion (`createFamily`)

Example 2 (Column family generation): The conversion of the entity `A` occurs seamlessly creating the homonym column family and its attributes. Next, when converting `B`, it searches for parent relationships and finds a total disjoint generalization. As the parent entity `A` is already converted, it creates the column family `B` and decides for a shared key. The same occurs for `C` and `D`.

E. Shared Key Generation

The algorithm in Figure 7 is responsible to generate a shared key. It receives as input an entity or relationship and returns the set of attributes that compose its key. Initially, the result set receives the key of the input$^{:1}$. If such a key does not exist$^{:2}$, then if the input is an entity$^{:3}$, it builds a list of generalization or mandatory `1:L` or `1:1` relationships, respectively$^{:4}$, so the result set receives the first item on the list$^{:5}$. We chose the first item in order to get the parent relationship with the most potential cardinality, so the shared key concept can be better exploited. If the input is a relationship$^{:6}$, the key is the side of the relationship with maximum and minimum cardinalities as 1$^{:7}$. Finally, if the result set is still empty$^{:9}$, a customized key is defined as a new column ID$^{:10}$.

```
   Input: Entity or relationship ($\epsilon$)
   Output: Attributes which compose the key ($\omega$)
1  $\omega \leftarrow \epsilon.Key$
2  if $\omega = \emptyset$ then
3      if $\epsilon$ is entity then
4          $\pi \leftarrow$ Parent generalizations, 1:L and 1:1 parent mandatory
             relationships of $\epsilon$, in this order
5          $\omega \leftarrow \pi_1.Key$
6      else
7          $\omega \leftarrow$ key of the (1,1) side of $\epsilon$, if exists
8      end
9      if $\omega = \emptyset$ then
10         $\omega \leftarrow \{ID\}$
11     end
12 end
13 return $\omega$
```

Figure 7. Algorithm for key definition (`defineKey`)

Example 3 (Shared key generation): Consider the initial conversion of entity `A` of Figure 1. As it has an identifier attribute (`id1`), it is defined as the key of its corresponding column family. As `A` is a parent entity of `B`, `C`, `D`, `R1` and `R3`, they all share its key, as shown in Figure 5.

F. Artificial Relation Generation

When a shared key is not possible, e.g., for the conversion of `N:M` relationships and partial generalizations, an artificial relation is defined. The term artificial stands for the fact that it is a relation whose integrity must be managed by the application, not the DBMS. The algorithm in Figure 8 is invoked by the algorithm in Figure 10, but we present it close to the definition of shared keys for sake of better understanding. The algorithm in Figure 8 is responsible to define artificial relationships in a columnar DB logical schema through the generation of additional column families for them. These additional column families are categorized as *auxiliary* (when it shares the key with its parent) or *intermediary* (when it is an independent family referenced by an auxiliary one).

```
Input: Two column familes ($\epsilon_1; \epsilon_2$) and the relationship ($\pi$)
Output: Additional column families ($\omega$)
1  $\omega \leftarrow \emptyset$
2  if $\epsilon_1$ was created for a relationship then
3      $\epsilon' \leftarrow \epsilon_1$
4  else
5      $\epsilon' \leftarrow$ Preexistent column family between $\epsilon_1$ and $\epsilon_2$
6  end
7  if $\epsilon' = \emptyset$ then
8      $\epsilon_T \leftarrow$ Temporary entity whose name is composed by the names
         of the input associated with $\epsilon_1$ through a $1:1$ relationship
9      $\epsilon' \leftarrow createFamily(\epsilon_T)$
10     $\omega \leftarrow \epsilon'$
11 end
12 $\delta_1 \leftarrow$ New key column
13 $\delta_1.Name \leftarrow \epsilon_2.Name$
14 $\epsilon' \leftarrow \epsilon' \cup \delta_1$
15 if $\pi.Type \neq (N:M) \vee \pi$ promoted to associative entity then
16     $\delta_2 \leftarrow$ New column that represent an artificial relation
17     $\delta_2.Name \leftarrow \epsilon_2.Name$
18     $\epsilon_2 \leftarrow \epsilon_2 \cup \delta_2$
19 end
20 return $\omega$
```

Figure 8. Algorithm for artificial relation creation (`createArtificialRelation`)

The input of this algorithm are two column families (first and second ones), and the relationship. It outputs additional column families necessary to the definition of the artificial relation. The algorithm is divided in two parts: *(i)* the definition of the source column family, and *(ii)* the definition of the artificial relation.

In order to define the source column family, it checks if the first column family was created from a relationship$^{:2}$. If yes, it means that it can behave as an auxiliary family that can receive additional columns (Example 4), so it is set as the source column family$^{:3}$. If not$^{:4}$, the algorithm searches for a column family that was created to associate the two input column families (Example 9), and sets it as the source family$^{:5}$. After that, if no source family was defined$^{:7}$, a temporary entity is created$^{:8}$ with a `1:1` relationship with the entity referring the first column family (to allow the definition of a shared key), and the result of the conversion of the temporary entity is set as the source column family (Example 5).

After the source column family is defined, the artificial relation is established by creating a new key column$^{:12}$, naming it$^{:13}$ and adding it to the source column family$^{:14}$. Only if the relationship is not `N:M` or it is promoted to an associative entity$^{:15}$, the other side of the artificial relation is defined as a column$^{:16}$. It is named$^{:17}$ and added to the second column family$^{:18}$. At the end, the generated column family set is returned.

Example 4 (Artificial relation for N-ary relationship): A `N:M` relationship promoted to an associative entity is handled exactly like a N-ary relationship. When the entity `E` of Figure 1 is going to be converted, it is detected that it has a `N:M` relationship (`R3`) with `H` that was promoted to an associative entity. So, it creates the column family `R3`. The definition of the key detects that there is a parent `1:1` relationship with `D` through `R2` and shares its key. Then, an artificial relation is defined from `E` to `H` through `R3` (column `E`). When `H` is further converted, the artificial relation for the other direction is created (column `H` in `R3` - see Figure 5).

Example 5 (Artificial relation for 1:N relationship): When the conversion process analyzes entity `E`, it detects that it has one parent entity `B` through `R1`. So, `B` is converted first. After, the column family `E` is created and its relationships are converted, in this case, `R1`. During `R1` conversion (Algorithm in Figure 10), it detects that it is a binary `1:N` relationship with attributes. Therefore, a column family is created for the relationship. When it happens, the algorithm detects that it can share a key with its mandatory side. So, the column family `R1` is created and it receives the key of `E` as a secondary key. In this way, `B` can reach any `E`. Finally, `E` receives a column referencing `R1` and the association becomes bidirectional.

G. Column Generation

The algorithm in Figure 9 is responsible to convert an attribute of an entity or relationship. It receives as input the target column family and the attribute to be converted. The output is a set of additional column families. The first part verifies if the attribute is composite$^{:2}$. If yes, it creates a new column family$^{:3}$, names it$^{:4}$, gets the key of the input column family$^{:5}$ (shared key), sets the same cardinality from the conceptual attribute$^{:6}$ and then, for each child attribute of the composite attribute$^{:7}$, it calls itself recursively$^{:8}$ to convert it, and the output is added to the result set. Next, the created column family is added to the result set$^{:10}$. If the attribute is not composite$^{:11}$, a new column is created$^{:12}$, named$^{:13}$, the cardinality of the attribute is copied$^{:14}$ and the column is added to the input column family$^{:15}$. At the end, the result set with the possible created column families is returned$^{:17}$.

Based on the algorithm in Figure 9, we can summarize the attribute conversion cases as follows: *(i)* a **key attribute** generates a column family key, i.e., a mandatory and unique information within a column family; *(ii)* a **mandatory and optional attribute** generates a column in a column family; *(iii)* a **multivalued attribute** generates a collection column in a column family; *(iv)* a **composite attribute**, as it is not a native feature of columnar DBs, it is represented as a new column family with a shared key; *(v)* a **multivalued composite attribute** is converted in the same way of a composite attribute with the additional definition of a cardinality constraint for the generated column family.

Example 6 (Column generation): The attribute `a1` of the entity `A` is an example of a monovalued mandatory attribute. It generates a simple column in `A`. The column family `E-a2` is an example of how a composite attribute is converted. The composite attribute itself turns into a column family and their attributes become columns.

H. Relationship Conversion

This section details the conversion of EER relationships to a columnar DB logical schema. In traditional relational DB design, the conversion of `1:1` relationships usually merges the involved entities into an unique component in the logical schema [10][11]. Instead, our approach creates at least two

```
Input: Column family ($\epsilon'$) and attribute ($\delta$) to convert
Output: Additional column families ($\omega$)
$\omega \leftarrow \emptyset$
if $\delta$ is composite then
    $\epsilon'' \leftarrow$ New empty column family
    $\epsilon''.Name \leftarrow \epsilon'.Name + \delta.Name$
    $\epsilon''.Key \leftarrow \epsilon'.Key$
    $\epsilon''.Cardinality \leftarrow \delta.Cardinality$
    foreach $\delta' \in \delta | \delta'$ is a child attribute do
        $\omega \leftarrow \omega \cup convertAttribute(\epsilon'', \delta')$
    end
    $\omega \leftarrow \omega \cup \epsilon''$
else
    $\delta' \leftarrow$ New column
    $\delta'.Name \leftarrow \delta.Name$
    $\delta'.Cardinality \leftarrow \delta.Cardinality$
    $\epsilon' \leftarrow \epsilon' \cup \delta'$
end
return $\omega$
```

Figure 9. Algorithm for attribute to column conversion (`convertAttribute`)

column families (a third one is additionally created if the relationship has attributes) and typically a shared key is used to nest it. The advantage of this approach is that the shared key puts data together through sharding. The merging of entities can leverage underutilization as all the columns of a key are loaded to memory.

Different from `1:1`, `1:N` relationships always generate artificial relations. It is also defined an auxiliary column family associated with the parent side through a shared key to handle relationship information. This separation allows the auxiliary entity to concentrate reads because it knows the exact keys of related data and the application can conveniently decide which information is important to acquire.

For `N:M` relationships, two difficulties arise: *(i)* how to access the data of the related entity through the relationship, and *(ii)* how to do it efficiently. For a better adherence to columnar DBs, it is necessary to both column families to know each other keys. Thus, instead of including the relationship in a particular column family, it is created an auxiliary one on each column family that refers each other in order to maintain bidirectional navigability, separation of concerns and keeping related data near.This strategy is expanded to N-ary relationships, where the main difference is that it has more dimensions associated to it and the creation of the relationship column family is mandatory to hold all the references and its different cardinalities.

Example 7 (Binary relationship conversion): The relationship `R1` is `1:N`. In this case, an artificial relation is created. We have a parent column family `B` and a child `E`. The parent receives an auxiliary column family `R1` with a shared key that points to the child and the attributes of the relationship. The child receives a column referring the parent. So, it is possible to navigate to `B` children through `R1`, and `E` can access its parent through the new column.

The conversion of relationships is supported by the algorithm in Figure 10. It deals with all existing EER relationship types. Its input is a relationship and the source entity, and its output is a column family set. In the first part of the algorithm, we initialize the result set to empty$^{:1}$ and proceed the analysis and treatment of each type of relationship$^{:2}$:

- **Binary or reflexive** relationship does not consider a promoted associative entity or `N:M` relationship$^{:3}$. If the relationship has attributes$^{:4}$, it creates a column family$^{:5}$ and makes it the parent of the relationship$^{:6}$. Then, if the parent and the source column family do not share the key or the relationship is reflexive$^{:8}$, the creation of an artificial relation is triggered$^{:9}$ and the output is added to the result set;
- **N-ary** relationship considers a promoted associative entity or `N:M` relationship$^{:12}$. It creates a column family to the relationship$^{:13}$ and a temporary binary relationship (with the same cardinality on the source entity, and maximum cardinality 1 on the output column family$^{:14}$) that is converted recursively. Its output is added the result set$^{:15}$;
- **Generalization or union** relationship initially checks if it is partial or the entity do not share a key with its parent$^{:18}$. If so, an inverted artificial relation between them is created, using the source as parent and the parent entity in the relationship as child$^{:19}$. The generated output is added to the result set.

Temporary relationships are transient and its lifetime ends within its scope. So, it ceases to exist after its use. Its objective is to break the input relationship into smaller binary ones that can be handled by the available structures in our notation for columnar DBs.

```
Input: Relationship ($\pi$) and source column family ($\epsilon$)
Output: Column family set ($\omega$)
1  $\omega \leftarrow \emptyset$
2  switch $\pi.Type$ do
3      case (Binary $\vee$ Reflexive) $\wedge$ not (N:M $\wedge$ promoted to associative
         entity) do
4          if $\pi.Attributes \neq \emptyset$ then
5              $\omega \leftarrow createFamily(\pi)$
6              $\pi.Parent \leftarrow \omega$
7          end
8          if $\epsilon.Key \neq \pi.Parent.Key \vee \pi.Type = Reflexive$ then
9              $\omega \leftarrow \omega \cup createArtificialRelation(\pi.Parent, \epsilon, \pi)$
10         end
11     end
12     case N-ary $\vee$ (N:M $\wedge$ promoted to associative entity) do
13         $\epsilon_R \leftarrow createFamily(\pi)$
14         $\pi_T \leftarrow$ Temporary binary relationship with the same
             cardinality on $\epsilon$ and maximum equals 1 on $\epsilon_R$
15         $\omega \leftarrow \omega \cup \epsilon_R \cup convertRelationship(\pi_T, \epsilon)$
16     end
17     case Generalization $\vee$ Union do
18         if $\pi$ is partial or any entity in $\pi$ cannot share key among
             them then
19             $\omega \leftarrow \omega \cup createArtificialRelation(\epsilon, \pi.Parent, \pi)$
20         end
21     end
22 end
23 return $\omega$
```

Figure 10. Algorithm for relationship convertion (`convertRelationship`)

Example 8 (Generalization conversion): Entity `A` is the parent entity in the generalization relationship and it is converted before its specializations. For entity `B`, the algorithm verifies that the parent entity (`A`) is already converted and then proceeds its own conversion. So, the family `B` is created and, as it is a specialized entity in a total generalization, it shares its parent key. In this case, no new column family is created as the relationship is represented by the shared key. The same reasoning is applied to the entities `C` and `D`.

Total unions are similar to total and disjoint generalizations (t,d) because all instances have a single inheritance. Thus, the same reasoning to convert generalizations applies. Partial unions are more complex because of the possibility of multiple inheritance, as well as none at all. Thus, the conversion process treats this kind of relation as a N-ary relation.

Example 9 (Union conversion): The entity `H` is a partial union of `F` and `G`. Considering that the first entity to be converted is `F`, the conversion process initially checks if `F` parents were already converted, as explained before. In this

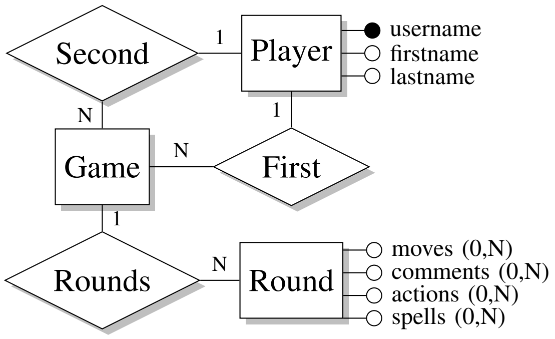
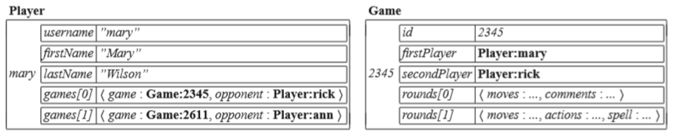


Figure 11. EER schema obtained through a reverse engineering process from the logical modeling for a game domain presented in [12]



Player

mary	*username*	*"mary"*
	firstName	*"Mary"*
	lastName	*"Wilson"*
	games[0]	⟨ *game* : **Game:2345**, *opponent* : **Player:rick** ⟩
	games[1]	⟨ *game* : **Game:2611**, *opponent* : **Player:ann** ⟩

Game

2345	*id*	*2345*
	firstPlayer	**Player:mary**
	secondPlayer	**Player:rick**
	rounds[0]	⟨ *moves* : ..., *comments* : ... ⟩
	rounds[1]	⟨ *moves* : ..., *actions* : ..., *spell* : ... ⟩

Figure 12. A sample in the NoAM abstract model [12].

case, H was not treated yet, and then H is converted first. The H column family is created and, as it does not have any parent relation, the processing returns to F, which is then converted. At this time, it is verified that the union is partial and an intermediary column family to hold the artificial relation is created (F−H). The same holds to G.

V. Experimental Evaluation

The aggregation strategy for logical modeling prioritizes data accessing, avoiding read and write operations on different nodes. This is the reasoning behind *NoAM* [12], a close related work that also deals with NoSQL logical design. Such a strategy considers that is most efficient to gather all related data in a single operation. However, some problems arise from this strategy, like transporting irrelevant data for query operations or the need to persist the whole aggregate for update operations. This experiment intends to explore these limitations and to highlight our approach as a more efficient solution. The case study proposed by *NoAM* is a game application, and we compare it with our approach by adapting their experiments to a scenario which game data grows to a deterrent size. This scenario is suitable to modern game applications that, in many cases, simply never end.

To evaluate our approach, the EER schema in Figure 11 was generated through a reverse engineering process from the NoAM logical modeling (Figure 12), composed by two aggregates: Player (with game references) and Game (with its rounds).

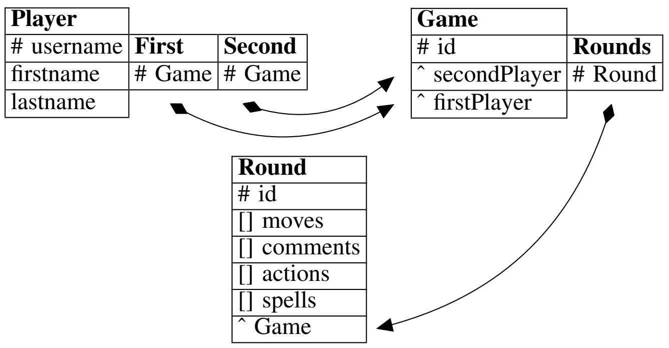
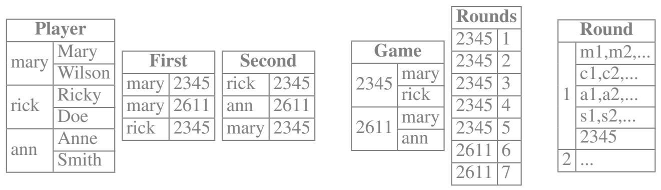


Figure 13. Logical modeling for a columnar DB in the game domain generated by our proposal.



Player	
mary	Mary
	Wilson
rick	Ricky
	Doe
ann	Anne
	Smith

First	
mary	2345
mary	2611
rick	2345

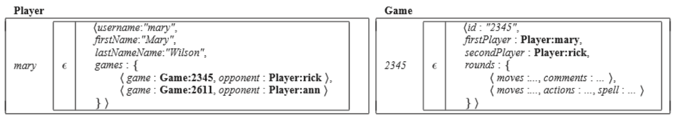
Second	
rick	2345
ann	2611
mary	2345

Game	
2345	mary
	rick
2611	mary
	ann

Rounds	
2345	1
2345	2
2345	3
2345	4
2345	5
2611	6
2611	7

Round	
1	m1,m2,...
	c1,c2,...
	a1,a2,...
	s1,s2,...
	2345
2	...

Figure 14. Physical modeling sample for in the game domain generated by our proposal.



Player

mary	ϵ	⟨*username*:"*mary*", *firstName*:"*Mary*", *lastNameName*:"*Wilson*", *games* : { ⟨ *game* : **Game:2345**, *opponent* : **Player:rick** ⟩, ⟨ *game* : **Game:2611**, *opponent* : **Player:ann** ⟩ } ⟩

Game

2345	ϵ	⟨*id* : "*2345*", *firstPlayer* : **Player:mary**, *secondPlayer* : **Player:rick**, *rounds* : { ⟨ *moves* :..., *comments* : ... ⟩, ⟨ *moves* :..., *actions* : ..., *spell* : ... ⟩ } ⟩

Figure 15. Physical modeling sample for in the game domain in NoAM [12].

The EER schema is then converted by our approach into the columnar logical schema presented in Figure 13. A sample of persisted data generated by our approach is shown in Figure 14, and part of this same sample represented by the NoAM approach is shown in Figure 15. The main difference between both proposals is that ours expands the number of column families from two to six. However, three of them are referenced by shared keys (First, Second and Rounds), so the data is near and easily referenced. Three artificial relations exist and the same number of references are made by the NoAM proposal (firstPlayer, secondPlayer and Game). Besides, all the attributes in both modelings are similar, except for opponent, that it is assumed to be the opposite player.

In order to evaluate our proposal, we have implemented both logical schemata in the Cassandra columnar DB, and we compare read/write operation timespan according to a scenario which we believe that the NoAM approach cannot handle well. For running our experiments, a remote Cassandra cluster with three nodes was deployed. Its overall performance is not the focus of this work nor the latency with data transport.

An algorithm to rule the experiment was defined and consists of two parts: *(i)* creating a game and; *(ii)* playing the game. In short, for each created game, a hundred rounds multiplied by the game count were created. Each game has two players and both are updated to maintain a list of games they are playing. A game creation is a single write operation. To update the list of games of both players, 2 reads and 2 writes are necessary. To add a round to a game, it is needed 1 read and 1 write operation. Thus, at the end of the fourth game, 2.020 reads and writes were accomplished.

The presented charts are comparisons between NoAM (solid line) and our proposal (dotted line). The X axis represents the iterations of the algorithm, and the Y axis is the average spent time in seconds. Figure 16 shows the spent time with write operations for the four games and its rounds, and Figure 17 shows the spent time with read operations. The deep drops in these charts (near iterations 100, 300, 600 and 1000) denotes the start of new games (with empty rounds).

The experiment shows that, as the data blocks (or aggregates) grows, in NoAM approach, the timespan also raises. For small sized aggregates, the timing is similar for both approaches. However, when a single game reaches 70 rounds, they have a drop of 50% in terms of performance comparing to our approach. With a hundred rounds, NoAM is 2.6 times slower. Therefore, the increasing size of the aggregate impacts

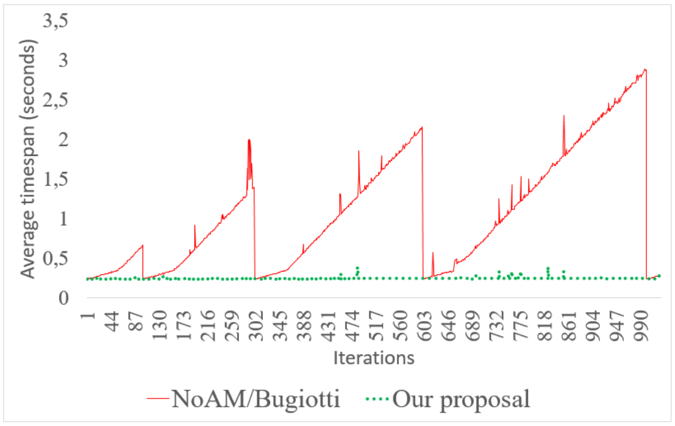


Figure 16. Average write time for a thousand iterations.

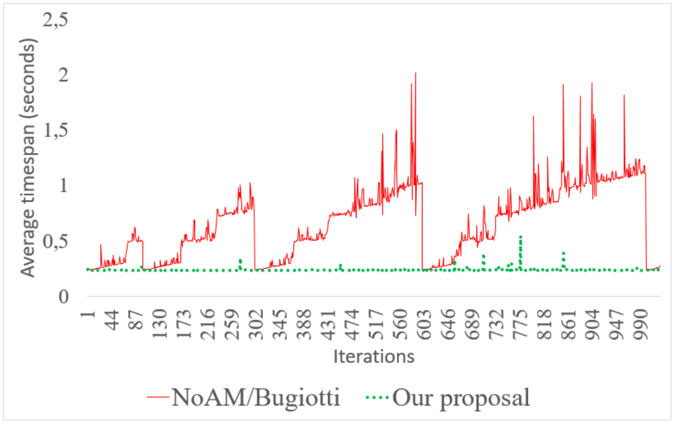


Figure 17. Average read time for a thousand iterations.

almost linearly to their performance decrease. This situation does not occur in our proposal because we make use of significantly smaller amount of data for each operation, and it happens independently of the hierarchical height on Y axis. Thus, our approach continues to be scalable as the number of game rounds grows.

The only inconvenient is that to gather all game data, our approach needs to pose several queries. However, these round-trips have a minimized impact on performance, as recent Application Programming Interfaces (API) can issue several queries in a single request to the server.

VI. Conclusion

This work represents a connection between classical DB design and columnar DBs, proposing an efficient approach for DB columnar logical design from an EER conceptual schema. Our contributions are a logical notation for columnar DBs, a set of conversion algorithms that generates a logical schema in that notation, as well as an experimental evaluation that compares our approach against a close related work (the *NoAM* approach), with very promising results. Our logical notation defines a minimal set of concepts needed to achieve a suitable structure to be implemented in a columnar DB.

The experimental evaluation shows a data modeling for columnar DB which reveals to be impracticable to NoAM [12], but viable to our approach.

We argue that scaled and massive data is not only for data mining. This work makes a progress in an area that urges to make NoSQL a reliable alternative to classical relational DB design. Domains where data that tends to grow very fast require efficient logical modeling strategies, as proposed in this paper.

Future work include experiments with existing benchmarks and other typical Big Data domains, like social networks, as well as the consideration of the application workload information in our logical design process. Workload information is important as a guide to define optimized logical structures for the most frequently accessed data by the application operations.

References

[1] C. Curino et al., “Relational cloud: A database service for the cloud,” in 5th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, January 2011, pp. 235–240.

[2] T. Hoff, “What the heck are you actually using nosql for?” http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html, 2010, retrieved: Apr, 2016.

[3] A. B. M. Moniruzzaman and S. A. Hossain, “Nosql database: New era of databases for big data analytics-classification, characteristics and comparison,” International Journal of Database Theory and Application, vol. 6, no. 4, 2013, pp. 1–14.

[4] K. Kaur and R. Rani, “Modeling and querying data in nosql databases,” in Big Data, 2013 IEEE International Conference on, Oct 2013, pp. 1–7.

[5] Solid IT, “Db–engines ranking,” http://db-engines.com/en/ranking, 2016, retrieved: Apr, 2016.

[6] P. J. Sadalage and M. Fowler, NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education, 2012.

[7] S. B. Navathe, C. Batini, and S. Ceri, “Conceptual database design: an entity-relationship approach,” Redwood City: Benjamin Cummings, 1992.

[8] G. Wang and J. Tang, “The nosql principles and basic application of cassandra model,” in Computer Science Service System (CSSS), 2012 on International Conference, Aug 2012, pp. 1332–1335.

[9] R. Elmasri and S. B. Navathe, Database systems. Pearson, 2005.

[10] R. Schroeder and R. d. S. Mello, “Improving query performance on xml documents: a workload-driven design approach,” in Proceedings of the eighth ACM symposium on Document engineering. ACM, 2008, pp. 177–186.

[11] J. Fong, “Mapping extended entity-relationship model to object modeling technique,” vol. 24, 1995, pp. 18–22.

[12] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone, “Database design for nosql systems,” in Conceptual Modeling. Springer, 2014, pp. 223–231.

[13] A. Chebotko, A. Kashlev, and S. Lu, “A big data modeling methodology for apache cassandra,” in Big Data (BigData Congress), 2015 IEEE International Congress on. IEEE, 2015, pp. 238–245.

[14] J. Sharp, D. McMurtry, A. Oakley, M. Subramanian, and H. Zhang, “Data access for highly-scalable solutions: Using sql, nosql, and polyglot persistence,” Microsoft patterns & practices, 2013.

[15] A. Schram and K. M. Anderson, “Mysql to nosql: data modeling challenges in supporting scalability,” in Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. ACM, 2012, pp. 191–202.

[16] E. Meijer and G. Bierman, “A co–relational model of data for large shared data banks,” Communications of the ACM, vol. 54, no. 4, 2011, pp. 49–58.

[17] L. Cabibbo, “Ondm: An object-nosql datastore mapper,” Faculty of Engineering, Roma Tre University. Retrieved June 15th, 2013.

[18] A. Milanović and M. Mijajlović, “A survey of post-relational data management and nosql movement,” Faculty of Mathematics University of Belgrade, Serbia, 2012.

[19] R. Mathies, “Cassandra data model,” http://wiki.apache.org/cassandra/DataModelv2, 2015, retrieved: Apr, 2016.

[20] Apache Foundation, “Cassandra wiki,” http://wiki.apache.org/cassandra, 2009, retrieved: Feb, 2015.

[21] R. Cattell, “Scalable sql and nosql data stores,” SIGMOD Record, vol. 39, no. 4, 2010, pp. 12–27.

[22] B. Schwartz, "Schemaless databases don't exist," https://vividcortex.com/blog/2015/02/24/schemaless-databases-dont-exist, 2015, retrieved: Apr, 2016.