# A Big Data Modeling Methodology for Apache Cassandra

Artem Chebotko
DataStax Inc.
Email: achebotko@datastax.com

Andrey Kashlev
Wayne State University
Email: andrey.kashlev@wayne.edu

Shiyong Lu
Wayne State University
Email: shiyong@wayne.edu

***Abstract*—Apache Cassandra is a leading distributed database of choice when it comes to big data management with zero downtime, linear scalability, and seamless multiple data center deployment. With increasingly wider adoption of Cassandra for online transaction processing by hundreds of Web-scale companies, there is a growing need for a rigorous and practical data modeling approach that ensures sound and efficient schema design. This work i) proposes the first query-driven big data modeling methodology for Apache Cassandra, ii) defines important data modeling principles, mapping rules, and mapping patterns to guide logical data modeling, iii) presents visual diagrams for Cassandra logical and physical data models, and iv) demonstrates a data modeling tool that automates the entire data modeling process.**

***Keywords*—Apache Cassandra, data modeling, automation, KDM, database design, big data, Chebotko Diagrams, CQL**

## I. Introduction

Apache Cassandra [1], [2] is a leading transactional, scalable, and highly-available distributed database. It is known to manage some of the world's largest datasets on clusters with many thousands of nodes deployed across multiple data centers. Cassandra data management use cases include product catalogs and playlists, sensor data and Internet of Things, messaging and social networking, recommendation, personalization, fraud detection, and numerous other applications that deal with time series data. The wide adoption of Cassandra [3] in big data applications is attributed to, among other things, its scalable and fault-tolerant peer-to-peer architecture [4], versatile and flexible data model that evolved from the BigTable data model [5], declarative and user-friendly Cassandra Query Language (CQL), and very efficient write and read access paths that enable critical big data applications to stay always on, scale to millions of transactions per second, and handle node and even entire data center failures with ease. One of the biggest challenges that new projects face when adopting Cassandra is data modeling that has significant differences from traditional data modeling approaches used in the past.

Traditional data modeling methodology, which is used in relational databases, defines well-established steps shaped by decades of database research [6], [7], [8]. A database designer typically follows the database schema design workflow depicted in Fig. 1(a) to define a conceptual data model, map it to a relational data model, normalize relations, and apply various optimizations to produce an efficient database schema with tables and indexes. In this process, the primary focus is placed on understanding and organizing data into relations, minimizing data redundancy and avoiding data duplication. Queries play a secondary role in schema design. Query analysis is frequently omitted at the early design stage because of the expressivity of the Structured Query Language (SQL) that readily supports relational joins, nested queries, data aggregation, and numerous other features that help to retrieve a desired subset of stored data. As a result, traditional data modeling is a purely data-driven process, where data access patterns are only taken into account to create additional indexes and occasional materialized views to optimize the most frequently executed queries.

In contrast, known principles used in traditional database design cannot be directly applied to data modeling in Cassandra. First, the Cassandra data model is designed to achieve superior write and read performance for a specified set of queries that an application needs to run. Data modeling for Cassandra starts with application queries. Thus, designing Cassandra tables based on a conceptual data model alone, without taking queries into consideration, leads to either inefficient queries or queries that cannot be supported by a data model. Second, CQL does not support many of the constructs that are common in SQL, including expensive table joins and data aggregation. Instead, efficient Cassandra database schema design relies on data nesting or schema denormalization to enable complex queries to be answered by only accessing a single table. It is common that the same data is stored in multiple Cassandra tables to support different queries, which results in data duplication. Thus, the traditional philosophy of normalization and minimizing data redundancy is rather opposite to data modeling techniques for Cassandra. To summarize, traditional database design is not suitable for developing correct, let alone efficient Cassandra data models.

In this paper, we propose a novel query-driven data modeling methodology for Apache Cassandra. A high-level overview of our methodology is shown in Fig. 1(b). A Cassandra solution architect, a role that encompasses both database design and application design tasks, starts data modeling by building a conceptual data model and defining an application workflow to capture all application interactions with a database. The application workflow describes access patterns or queries that a data-driven application needs to run against the database. Based on the identified access patterns, the solution architect maps the conceptual data model to a logical data model. The logical data model specifies Cassandra tables that can efficiently support application queries according to the application workflow. Finally, additional physical optimizations concerning data types, keys, partition sizes, and ordering are applied to produce a physical data model that can be instantiated in Cassandra using CQL.

The most important innovation of our methodology, when compared to relational database design, is that the application workflow and the access patterns become first-class citizens

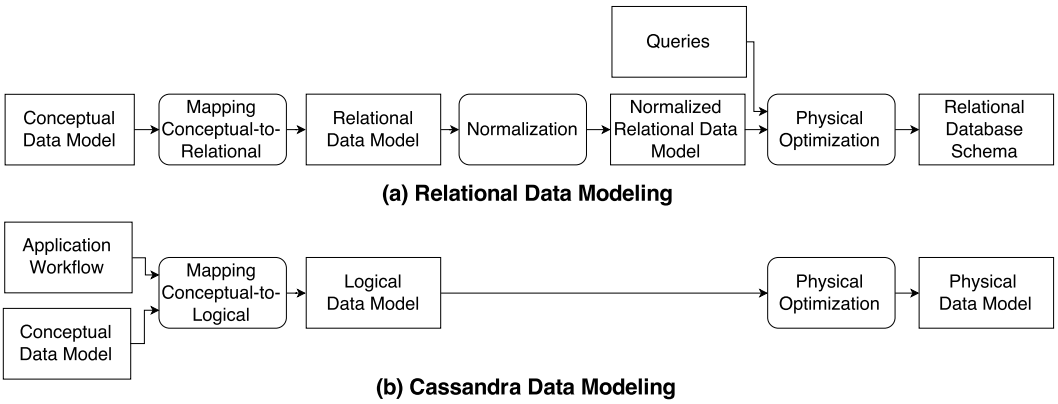



Fig. 1: Traditional data modeling compared with our proposed methodology for Cassandra.

in the data modeling process. Cassandra database design revolves around both the application workflow and the data, and both are of paramount importance. Another key difference of our approach compared to the traditional strategy is that normalization is eliminated and data nesting is used to design tables for the logical data model. This also implies that joins are replaced with data duplication and materialized views for complex application queries. These drastic differences demand much more than a mere adjustment of the data modeling practices. They call for a new way of thinking, a paradigm shift from purely data-driven approach to query-driven data modeling process.

To our best knowledge, this work presents the first query-driven data modeling methodology for Apache Cassandra. Our main contributions are: (i) a first-of-its-kind data modeling methodology for Apache Cassandra, (ii) a set of modeling principles, mapping rules, and mapping patterns that guide a logical data modeling process, (iii) a visualization technique, called *Chebotko Diagrams*, for logical and physical data models, and (iv) a data modeling tool, called *KDM*, that automates Cassandra database schema design according to the proposed methodology. Our methodology has been successfully applied to real world use cases at a number of companies and is incorporated as part of the DataStax Cassandra training curriculum [9].

The rest of the paper is organized as follows. Section II provides a background on the Cassandra data model. Section III introduces conceptual data modeling and application workflows. Section IV elaborates on a query-driven mapping from a conceptual data model to a logical data model. Section V briefly introduces physical data modeling. Section VI illustrates the use of *Chebotko Diagrams* for visualizing logical and physical data models. Section VII presents our *KDM* tool to automate the data modeling process. Finally, Sections VIII and IX present related work and conclusions.

## II. The Cassandra Data Model

A database schema in Cassandra is represented by a keyspace that serves as a top-level namespace where all other data objects, such as tables, reside[1]. Within a keyspace, a set of CQL tables is defined to store and query data for a particular application. In this section, we discuss the table and query models used in Cassandra.

### A. Table Model

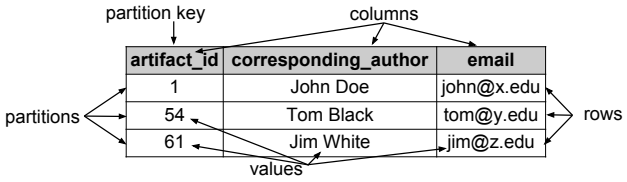
The notion of a table in Cassandra is different from the notion of a table in a relational database. A CQL table (hereafter referred to as a table) can be viewed as a set of partitions that contain rows with a similar structure. Each partition in a table has a unique partition key and each row in a partition may optionally have a unique clustering key. Both keys can be simple (one column) or composite (multiple columns). The combination of a partition key and a clustering key uniquely identifies a row in a table and is called a primary key. While the partition key component of a primary key is always mandatory, the clustering key component is optional. A table with no clustering key can only have single-row partitions because its primary key is equivalent to its partition key and there is a one-to-one mapping between partitions and rows. A table with a clustering key can have multi-row partitions because different rows in the same partition have different clustering keys. Rows in a multi-row partition are always ordered by clustering key values in ascending (default) or descending order.

A table schema defines a set of columns and a primary key. Each column is assigned a data type that can be primitive, such as *int* or *text*, or complex (*collection* data types), such as *set*, *list*, or *map*. A column may also be assigned a special *counter* data type, which is used to maintain a distributed counter that can be added to or subtracted from by concurrent transactions. In the presence of a counter column, all non-counter columns in a table must be part of the primary key. A column can be defined as *static*, which only makes sense in a table with multi-row partitions, to denote a column whose value is shared by all rows in a partition. Finally, a primary key is a sequence of columns consisting of partition key columns followed by optional clustering key columns. In CQL, partition key columns are delimited by additional parenthesis, which can be omitted if a partition key is simple. A primary key may not include counter, static, or collection columns.

To illustrate some of these notions, Fig. 2 shows two sample tables with CQL definitions and sample rows. In Fig. 2(a), the *Artifacts* table contains single-row partitions. Its primary key consists of one column *artifact_id* that is also a simple

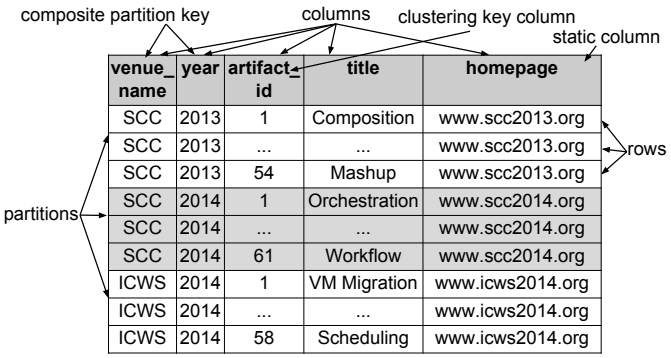
[1]Another important function of a keyspace is the specification of a data replication strategy, the topic that lies beyond the scope of this paper.

```
CREATE TABLE artifacts(
      artifact_id INT,
      corresponding_author TEXT,
      email TEXT,
      PRIMARY KEY (artifact_id));
```




**(a) Table *Artifacts* with single-row partitions**

```
CREATE TABLE artifacts_by_venue(
      venue_name TEXT,
      year INT,
      artifact_id INT,
      title TEXT,
      homepage TEXT STATIC,
      PRIMARY KEY ((venue_name,year),artifact_id));
```




**(b) Table *Artifacts_by_venue* with multi-row partitions**

Fig. 2: Sample tables in Cassandra.

partition key. This table is shown to have three single-row partitions. In Fig. 2(b), the *Artifacts_by_venue* table contains multi-row partitions. Its primary key consists of composite partition key (*venue_name*, *year*) and simple clustering key *artifact_id*. This table is shown to have three partitions, each one containing multiple rows. For any given partition, its rows are ordered by *artifact_id* in ascending order. In addition, *homepage* is defined as a static column, and therefore each partition can only have one *homepage* value that is shared by all the rows in that partition.

### B. Query Model

Queries over tables are expressed in CQL, which has an SQL-like syntax. Unlike SQL, CQL supports no binary operations, such as joins, and has a number of rules for query predicates that ensure efficiency and scalability:

- Only primary key columns may be used in a query predicate.
- All partition key columns must be restricted by values (i.e. equality search).
- All, some, or none of the clustering key columns can be used in a query predicate.
- If a clustering key column is used in a query predicate, then all clustering key columns that precede this clustering column in the primary key definition must also be used in the predicate.
- If a clustering key column is restricted by range (i.e. inequality search) in a query predicate, then all clustering key columns that precede this clustering column in the primary key definition must be restricted by values and no other clustering column can be used in the predicate.

Intuitively, a query that restricts all partition key columns by values returns all rows in a partition identified by the specified partition key. For example, the following query over the *Artifacts_by_venue* table in Fig. 2(b) returns all artifacts published in the venue *SCC 2013*:

```
SELECT artifact_id, title
FROM artifacts_by_venue
WHERE venue_name='SCC' AND year=2013
```

A query that restricts all partition key columns and some clustering key columns by values returns a subset of rows from a partition that satisfy such a predicate. Similarly, a query that restricts all partition key columns by values and one clustering key column by range (preceding clustering key columns are restricted by values) returns a subset of rows from a partition that satisfy such a predicate. For example, the following query over the *Artifacts_by_venue* table in Fig. 2(b) returns artifacts with id's from *1* to *20* published in *SCC 2013*:

```
SELECT artifact_id, title
FROM artifacts_by_venue
WHERE venue_name='SCC' AND year=2013 AND
artifact_id>=1 AND artifact_id<=20;
```

Query results are always ordered based on the default order specified for clustering key columns when a table is defined (the `CLUSTERING ORDER` construct), unless a query explicitly reverses the default order (the `ORDER BY` construct).

Finally, CQL supports a number of other features, such as queries that use secondary indexes, `IN`, and `ALLOW FILTERING` constructs. Our data modeling methodology does not directly rely on such queries as their performance is frequently unpredictable on large datasets. More details on the syntax and semantics of CQL can be found in [10].

## III. Conceptual Data Modeling and Application Workflow Modeling

The first step in the proposed methodology adds a whole new dimension to database design, not seen in the traditional relational approach. Designing a Cassandra database schema requires not only understanding of the to-be-managed data, but also understanding of how a data-driven application needs to access such data. The former is captured via a conceptual data model, such as an entity-relationship model. In particular, we choose to use Entity-Relationship Diagrams in Chen's notation [8] for conceptual data modeling because this notation is truly technology-independent and not tainted with any relational model features. The latter is captured via an application workflow diagram that defines data access patterns for individual application tasks. Each access pattern specifies what attributes to search for, search on, order by, or do aggregation on with a distributed counter. For readability, in this paper, we use verbal descriptions of access patterns. More formally, access patterns can be represented as graph queries written in a language similar to ERQL [11].

As a running example, we design a database for a digital library use case. The digital library features a collection of digital artifacts, such as papers and posters, which appeared in various venues. Registered users can leave feedback for venues and artifacts in the form of reviews, likes, and ratings. Fig. 3 shows a conceptual data model and an application workflow for our use case. The conceptual data model in Fig. 3(a) unambiguously defines all known entity types, relationship types, attribute types, key, cardinality, and other constraints. For example, a part of the diagram can be interpreted as "*user* is uniquely identified by *id* and may *post many reviews*, while each *review* is *posted* by *exactly one user*". The application workflow in Fig. 3(b) models a web-based application that allows users to interact with various web pages (tasks) to retrieve data using well-defined queries. For example, the uppermost task in the figure is the entry point to the application and allows searching for artifacts in a database based on one of the queries with different properties. As we show in the next section, both the conceptual data model and the application workflow have a profound effect on the design of a logical data model.

## IV. Logical Data Modeling

The crux of the Cassandra data modeling methodology is logical data modeling. It takes a conceptual data model and maps it to a logical data model based on queries defined in an application workflow. A logical data model corresponds to a Cassandra database schema with table schemas defining columns, primary, partition, and clustering keys. We define the query-driven conceptual-to-logical data model mapping via data modeling principles, mapping rules, and mapping patterns.

### A. Data Modeling Principles

The following four data modeling principles provide a foundation for the mapping of conceptual to logical data models.

*DMP1 (Know Your Data).* The first key to successful database design is understanding the data, which is captured with a conceptual data model. The importance and effort required for conceptual data modeling should not be underestimated. Entity, relationship, and attribute types on an ER diagram (e.g., see Fig. 3(a)) not only define which data pieces need to be stored in a database but also which data properties, such as entity type and relationship type keys, need to be preserved and relied on to organize data correctly.

For example, in Fig. 3(a), *name* and *year* constitute a venue key. This is based on our use case assumption that there cannot be two venues (e.g., conferences) with the same name that are held in the same year. If our assumption is false, the conceptual data model and overall design will have to change. Another example is the cardinality of the relationship type *features*. In this case, our use case assumption is that a venue can feature many artifacts and an artifact can only appear in one venue. Thus, given the one-to-many relationship type, the key of *features* is *id* of an artifact. Again, if our assumption is false, both the cardinalities and key will have to change, resulting in substantially different table schema design.

*DMP2 (Know Your Queries).* The second key to successful database design is queries, which are captured via an application workflow model. Like data, queries directly affect table schema design, and if our use case assumptions about the queries (e.g., see Fig. 3(b)) change, a database schema will have to change, too. In addition to considering queries and ensuring their correct support, we should also take into account an access path of each query to organize data efficiently.

We define the three broad access paths: 1) partition per query, 2) partition+ per query, and 3) table or table+ per query. The most efficient option is the "partition per query", when a query only retrieves one row, a subset of rows or all rows from a single partition. For example, both queries presented in Section II-B are examples of the "partition per query" access path. This access path should be the most common in an online transaction processing scenario but, in some cases, may not be possible or desirable (e.g., a partition may have to become very large to satisfy this path for a query). The "partition+ per query" and "table and table+ per query" paths refer to retrieving data from a few partitions in a table or from many partitions in one or more tables, respectively. While these access paths can be valid in some cases, they should be avoided to achieve optimal query performance.

*DMP3 (Data Nesting).* The third key to successful database design is data nesting. Data nesting refers to a technique that organizes multiple entities (usually of the same type) together based on a known criterion. Such criterion can be that all nested entities must have the same value for some attribute (e.g., venues with the same name) or that all nested entities must be related to a known entity of a different type (e.g., digital artifacts that appeared in a particular venue). Data nesting is used to achieve the "partition per query" access path, such that multiple nested entities can be retrieved from a single partition. There are two mechanisms in Cassandra to nest data: multi-row partitions and collection types. Our methodology primarily relies on multi-row partitions to achieve the best performance. For example, in Fig. 2(b), the *Artifacts_by_venue* table nests artifacts (rows) under venues (partitions) that featured those artifacts. In other words, each partition corresponds to a venue and each row in a given partition corresponds to an artifact that appeared in the partition venue. Tables with multi-row partitions are common in Cassandra databases.

*DMP4 (Data Duplication).* The fourth key to successful database design is data duplication. Duplicating data in Cassandra across multiple tables, partitions, and rows is a common practice that is required to efficiently support different queries over the same data. It is far better to duplicate data to enable the "partition per query" access path than to join data from multiple tables and partitions. For example, to support queries *Q1* and *Q2* in Fig. 3(b) via the efficient "partition per query" access path, we should create two separate tables that organize the same set of artifacts using different table primary keys. In the Cassandra world, the trade-off between space efficiency and time efficiency is almost always in favor of the latter.

### B. Mapping Rules

Based on the above data modeling principles, we define five mapping rules that guide a query-driven transition from a conceptual data model to a logical data model.

*MR1 (Entities and Relationships).* Entity and relationship types map to tables, while entities and relationships map to table rows. Attribute types that describe entities and relationships

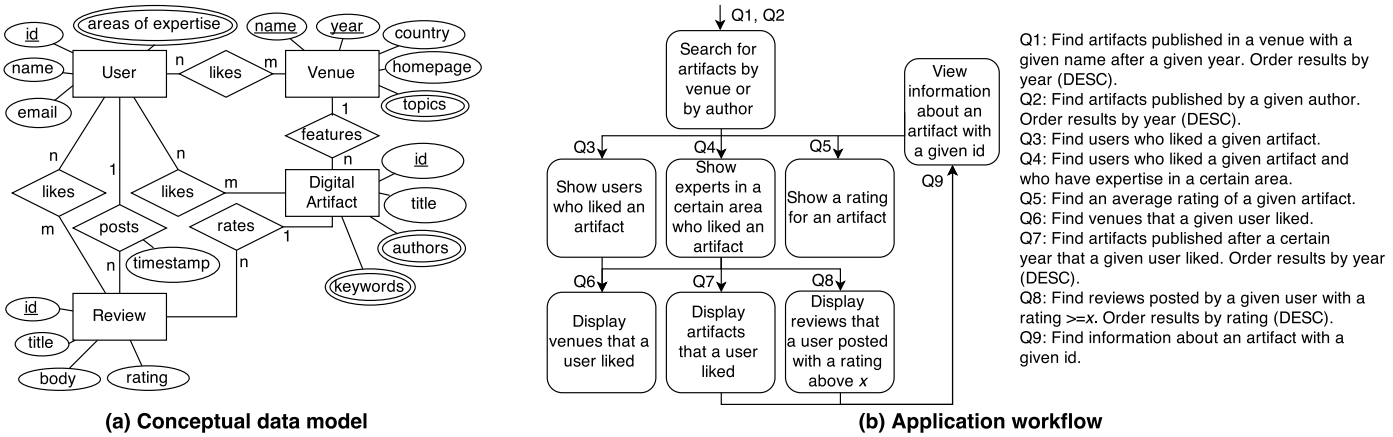



Fig. 3: A conceptual data model and an application workflow for the digital library use case.

at the conceptual level must be preserved as table columns at the logical level. Violation of this rule may lead to data loss.

*MR2 (Equality Search Attributes).* Equality search attributes, which are used in a query predicate, map to the prefix columns of a table primary key. Such columns must include all partition key columns and, optionally, one or more clustering key columns. Violation of this rule may result in inability to support query requirements.

*MR3 (Inequality Search Attributes).* An inequality search attribute, which is used in a query predicate, maps to a table clustering key column. In the primary key definition, a column that participates in inequality search must follow columns that participate in equality search. Violation of this rule may result in inability to support query requirements.

*MR4 (Ordering Attributes).* Ordering attributes, which are specified in a query, map to clustering key columns with ascending or descending clustering order as prescribed by the query. Violation of this rule may result in inability to support query requirements.

*MR5 (Key Attributes).* Key attribute types map to primary key columns. A table that stores entities or relationships as rows must include key attributes that uniquely identify these entities or relationships as part of the table primary key to uniquely identify table rows. Violation of this rule may lead to data loss.

To design a table schema, it is important to apply these mapping rules in the context of a particular query and a subgraph of the conceptual data model that the query deals with. The rules should be applied in the same order as they are listed above.

For example, Fig. 4 illustrates how the mapping rules are applied to design a table for query *Q1* (see Fig. 3(b)) that deals with the relationship *Venue-features-Digital Artifact* (see Fig. 3(a)). Fig. 4 visualizes a table resulting after each rule application using Chebotko's notation, where *K* and *C* denote partition and clustering key columns, respectively. The arrows next to the clustering key columns denote ascending (↑) or descending (↓) order. *MR1* results in table *Artifacts_by_venue* whose columns correspond to the attribute types used in the query to search for, search on, or order by. *MR2* maps

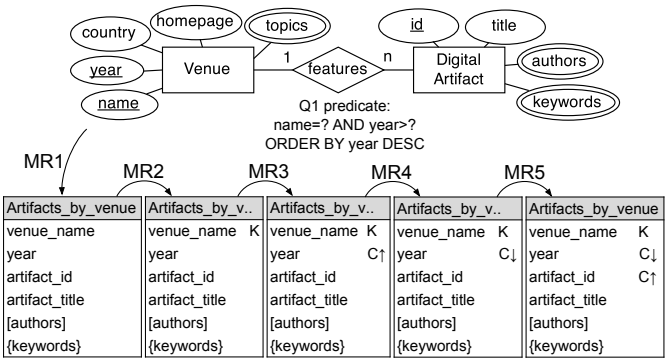


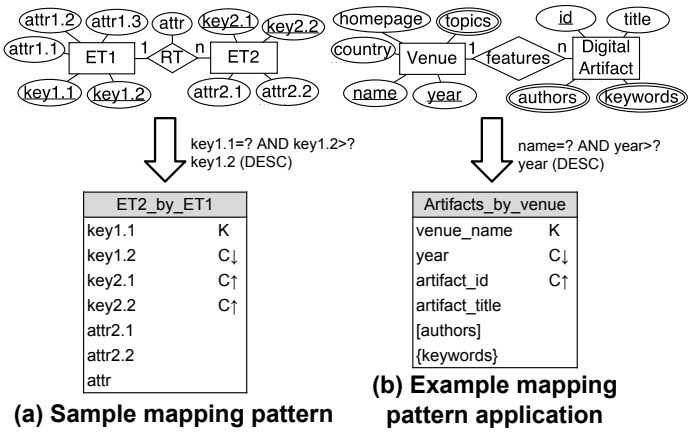

Fig. 4: Sample table schema design using the mapping rules.

the equality search attribute to the partition key column *venue_name*. *MR3* maps the inequality search attribute to the clustering key column *year*, and *MR4* changes the clustering order to descending. Finally, *MR5* maps the key attribute to the clustering key column *artifact_id*.

### C. Mapping Patterns

Based on the above mapping rules, we design mapping patterns that serve as the basis for automating Cassandra database schema design. Given a query and a conceptual data model subgraph that is relevant to the query, each mapping pattern defines final table schema design without the need to apply individual mapping rules. While we define a number of different mapping patterns [9], due to space limitations, we only present one mapping pattern and one example.

A sample mapping pattern is illustrated in Fig. 5(a). It is applicable for the case when a given query deals with one-to-many relationships and results in a table schema that nests many entities (rows) under one entity (partition) according to the relationships. When applied to query *Q1* (see Fig. 3(b)) and the relationship *Venue-features-Digital Artifact* (see Fig. 3(a)), this mapping pattern results in the table schema shown in Fig. 5(b). With our mapping patterns, logical data modeling becomes as simple as finding an appropriate mapping pattern and applying it, which can be automated.




**(a) Sample mapping pattern** **(b) Example mapping pattern application**

Fig. 5: A sample mapping pattern and the result of its application.

## V. Physical Data Modeling

The final step of our methodology is the analysis and optimization of a logical data model to produce a physical data model. While the modeling principles, mapping rules, and mapping patterns ensure a correct and efficient logical schema, there are additional efficiency concerns related to database engine constraints or finite cluster resources. A typical analysis of a logical data model involves the estimation of table partition sizes and data duplication factors. Some of the common optimization techniques include partition splitting, inverted indexes, data aggregation and concurrent data access optimizations. These and other techniques are described in [9].

## VI. Chebotko Diagrams

It is frequently useful to present logical and physical data model designs visually. To achieve this, we propose a novel visualization technique, called *Chebotko Diagram*, which presents a database schema design as a combination of individual table schemas and query-driven application workflow transitions. Some of the advantages of *Chebotko Diagrams*, when compared to regular CQL schema definition scripts, include improved overall readability, superior intelligibility for complex data models, and better expressivity featuring both table schemas and their supported application queries. Physical-level diagrams contain sufficient information to automatically generate a CQL script that instantiates a database schema, and can serve as reference documents for developers and architects that design and maintain a data-driven solution. The notation of *Chebotko Diagrams* is presented in Fig. 6.

Sample *Chebotko Diagrams* for the digital library use case are shown in Fig. 7. The logical-level diagram in Fig. 7(a) is derived from the conceptual data model and application workflow in Fig. 3 using the mapping rules and mapping patterns. The physical-level diagram in Fig. 7(b) is derived from the logical data model after specifying CQL data types for all columns and applying two minor optimizations: 1) a new column *avg_rating* is introduced into tables *Artifacts_by_venue*, *Artifacts_by_author*, and *Artifacts* to avoid an additional lookup in the *Ratings_by_artifact* table and 2) the *timestamp* column is eliminated from the *Reviews_by_user*

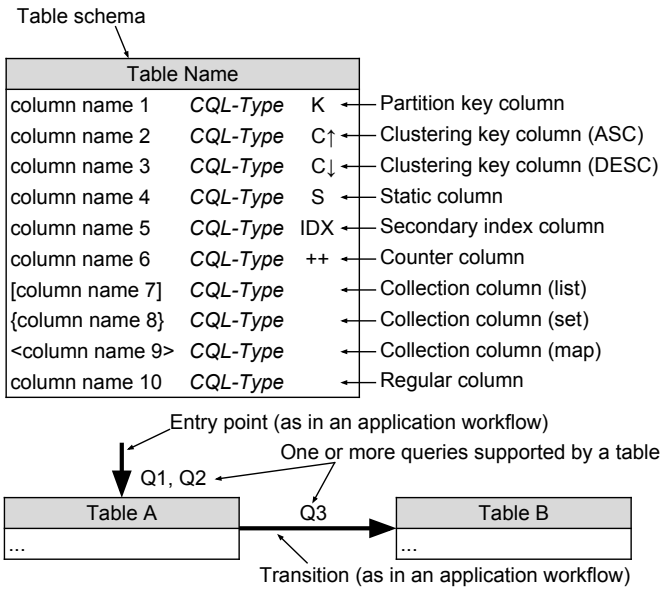



Fig. 6: The notation of Chebotko Diagrams.

table because a timestamp can be extracted from column *review_id* of type *TIMEUUID*.

## VII. Automation and the KDM Tool

To automate our proposed methodology in Fig. 1(b), we design and implement a Web-based data modeling tool, called *KDM*[2]. The tool relies on the mapping patterns and our proprietary algorithms to automate the most complex, error-prone, and time-consuming data modeling tasks: conceptual-to-logical mapping, logical-to-physical mapping, and CQL generation. *KDM*'s Cassandra data modeling automation workflow is shown in Fig. 8(a). Screenshots of *KDM*'s user interface corresponding to steps 1, 3, and 4 of this workflow are shown in Fig. 8(b).

Our tool was successfully validated for several use cases, including the digital library use case. Based on our experience, *KDM* can dramatically reduce time, streamline, and simplify the Cassandra database design process. *KDM* consistently generates sound and efficient data models, which is invaluable for less experienced users. For expert users, *KDM* supports a number of advanced features, such as automatic schema generation in the presence of type hierarchies, n-ary relationship types, explicit roles, and alternative keys.

## VIII. Related Work

Data modeling has always been a cornerstone of data management systems. Conceptual data modeling [8] and relational database design [6], [7] have been extensively studied and are now part of a typical database course. Unfortunately, the vast majority of relational data modeling techniques are not applicable to recently emerged big data (or *NoSQL*) management solutions. The need for new data modeling approaches for NoSQL databases has been widely recognized in both industry [12], [13] and academic [14], [15], [16] communities. Big data modeling is a challenging and open problem.

[2]KDM demo can be found at www.cs.wayne.edu/andrey/kdm

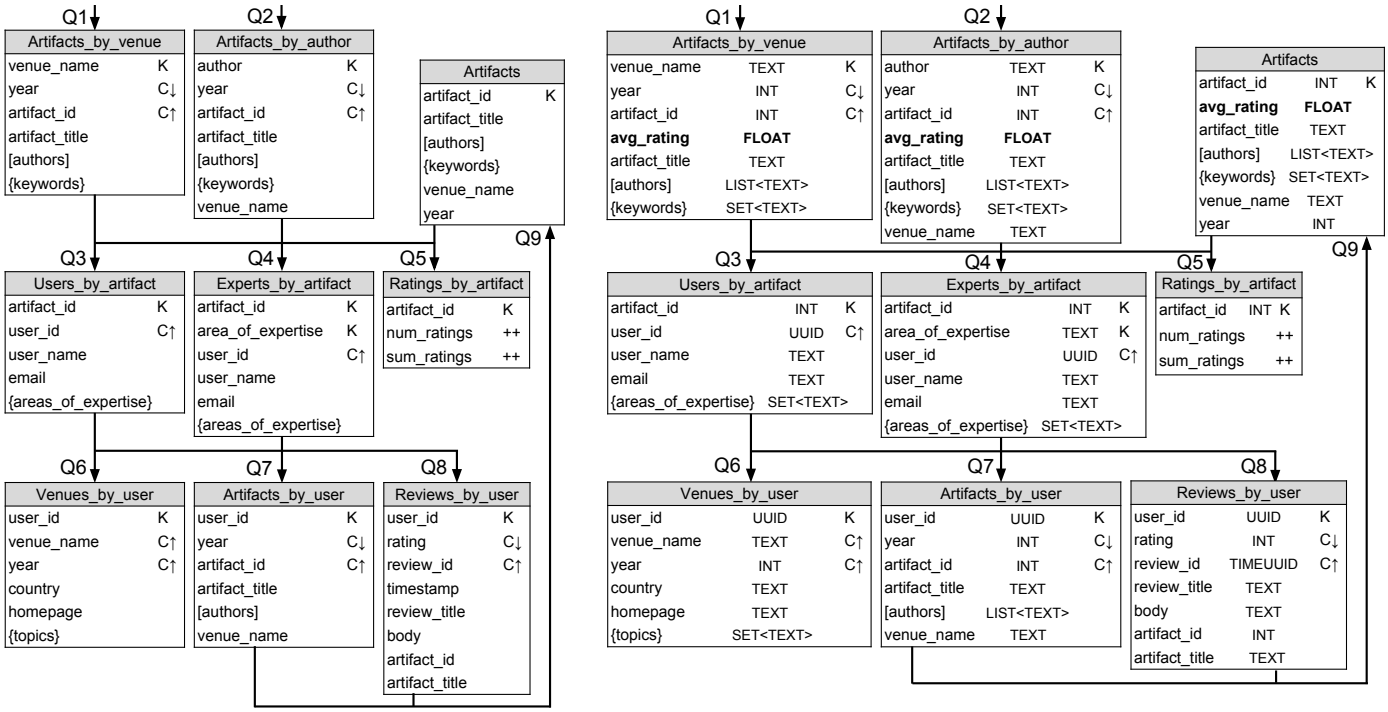



Fig. 7: Chebotko Diagrams for the digital library use case.

In the big data world, database systems are frequently classified into four broad categories [17] based on their data models: 1) key-value databases, such as *Riak* and *Redis*, 2) document databases, such as *Couchbase* and *MongoDB*, 3) column-family databases, such as *Cassandra* and *HBase*, and 4) graph databases, such as *Titan* and *Neo4J*. Key-value databases model data as key-value pairs. Document databases store JSON documents retrievable by keys. Column-family databases model data as table-like structures with multiple dimensions. Graph databases typically rely on internal ad-hoc data structures to store any graph data. An effort on a system-independent NoSQL database design is reported in [18], where the approach is based on *NoSQL Abstract Model* to specify an intermediate, system-independent data representation. Both our work and [18] recognize conceptual data modeling and query-driven design as essential activities of the data modeling process. While databases in different categories may share similar high-level data modeling ideas, such as data nesting (also, aggregation or embedding) or data duplication, many practical data modeling techniques rely on low-level features that are unique to a category and, more often, to a particular database.

In the Cassandra world, data modeling insights mostly appear in blog posts and presentations that focus on best practices, common use cases, and sample designs. Among some of the most helpful resources are *DataStax developer blog*[3], *DataStax data modeling page*[4], and *Patrick McFadin's presentations*[5]. To the best of our knowledge, this work is the first to propose a systematic and rigorous data modeling methodology for Apache Cassandra. *Chebotko Diagrams* for visualization and the *KDM* tool for automation are also novel and unique.

[3]http://www.datastax.com/dev/blog

[4]http://www.datastax.com/resources/data-modeling

[5]http://www.slideshare.net/patrickmcfadin

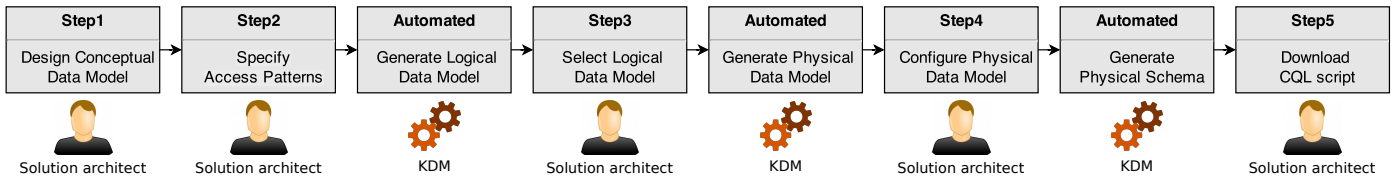
## IX. Conclusions and Future Work

In this paper, we introduced a rigorous query-driven data modeling methodology for Apache Cassandra. Our methodology was shown to be drastically different from the traditional relational data modeling approach in a number of ways, such as query-driven schema design, data nesting and data duplication. We elaborated on the fundamental data modeling principles for Cassandra, and defined mapping rules and mapping patterns to transition from technology-independent conceptual data models to Cassandra-specific logical data models. We also explained the role of physical data modeling and proposed a novel visualization technique, called *Chebotko Diagrams*, which can be used to capture complex logical and physical data models. Finally, we presented a powerful data modeling tool, called *KDM*, which automates some of the most complex, error-prone, and time-consuming data modeling tasks, including conceptual-to-logical mapping, logical-to-physical mapping, and CQL generation.

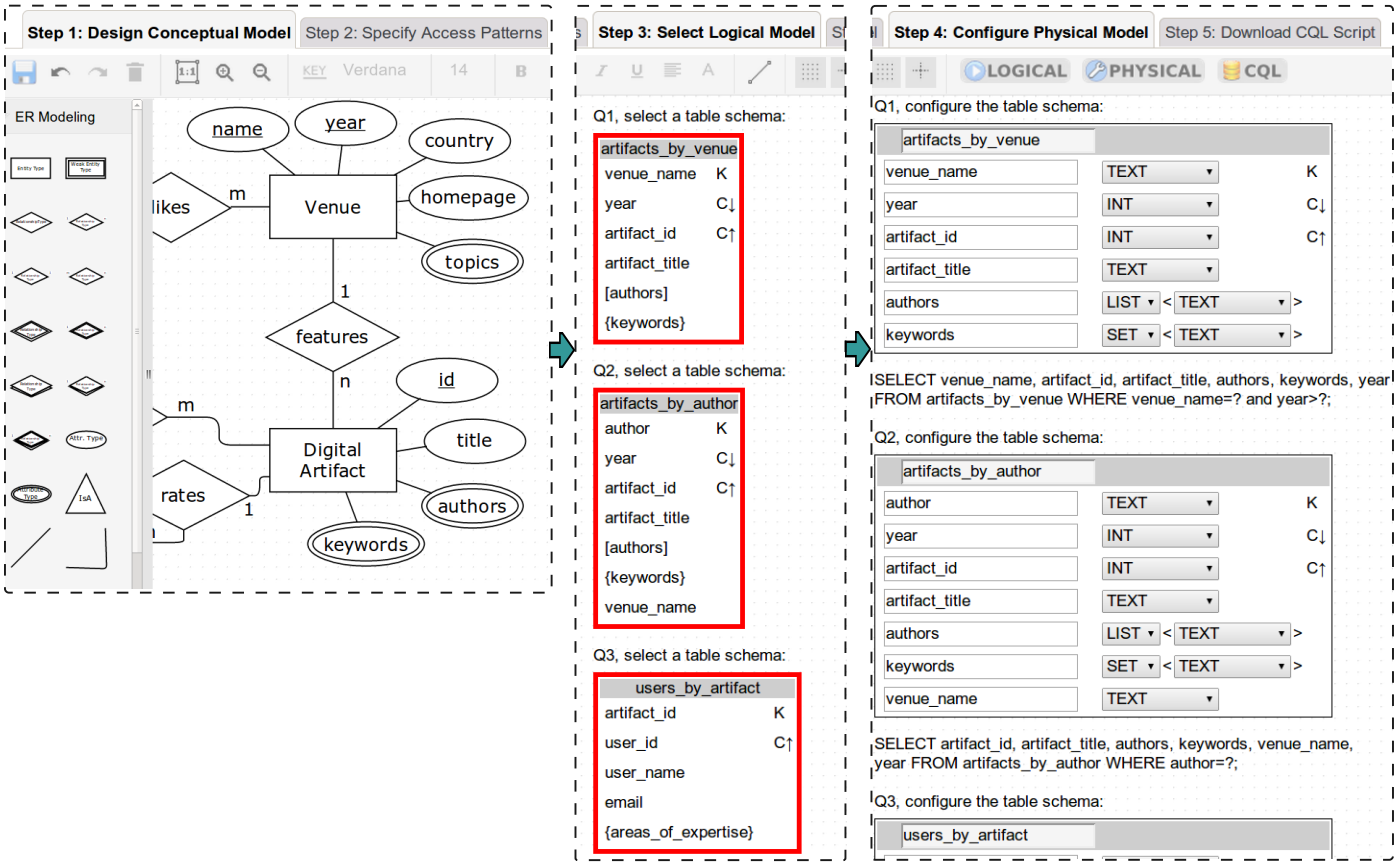
In the future, we plan to extend our work to support new Cassandra features, such as user defined data types and global indexes. We are also interested in exploring data modeling techniques in the context of analytic applications. Finally, we plan to explore schema evolution in Cassandra.

## Acknowledgements

Artem Chebotko would like to thank Anthony Piazza, Patrick McFadin, Jonathan Ellis, and Tim Berglund for their support at various stages of this effort.




**(a) KDM's Cassandra data modeling automation workflow**




**(b) Data modeling for the digial library use case performed in KDM.**

Fig. 8: Automated Cassandra data modeling using KDM.

## REFERENCES


[1] *Apache Cassandra Project*, http://cassandra.apache.org/.

[2] *Planet Cassandra*, http://http://planetcassandra.org/.

[3] *Companies that use Cassandra*, http://planetcassandra.org/companies/.

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

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[6] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[7] ——, "Further normalization of the data base relational model," *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

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

[9] *DataStax Cassandra Training Curriculum*, http://www.datastax.com/what-we-offer/products-services/training/apache-cassandra-data-modeling/.

[10] *Cassandra Query Language*, https://cassandra.apache.org/doc/cql3/CQL.html.

[11] M. Lawley and R. W. Topor, "A query language for EER schemas," in *Proceedings of the 5th Australasian Database Conference*, 1994, pp. 292–304.

[12] J. Maguire and P. O'Kelly, "Does data modeling still matter, amid the market shift to XML, NoSQL, big data, and cloud?" *White paper, https://www.embarcadero.com/phocadownload/new-papers/okelly-whitepaper-071513.pdf*, 2013.

[13] D. Hsieh, "NoSQL data modeling," *Ebay tech blog. http://www.ebaytechblog.com/2014/10/10/nosql-data-modeling*, 2014.

[14] A. Badia and D. Lemire, "A call to arms: revisiting database design," *SIGMOD Record*, vol. 40, no. 3, pp. 61–69, 2011.

[15] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone, "The relational model is dead, SQL is dead, and I don't feel so good myself," *SIGMOD Record*, vol. 42, no. 2, pp. 64–68, 2013.

[16] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han *et al.*, "Challenges and opportunities with big data - a community white paper developed by leading researchers across the United States," 2011.

[17] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[18] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone, "Database design for NoSQL systems," in *Proceedings of the 33rd International Conference on Conceptual Modeling*, 2014, pp. 223–231.