 **CLOUD**Graph™

CloudGraph Design Team  
TerraMeta Software, Inc.

Architecture Overview

**Revision History**

| **Revision** | **Date** | **Description** | **Author** |
| --- | --- | --- | --- |
| 1.0 | January 0, 2012 | CloudGraph architecture description | CloudGraph Design Team |

| Table of Contents |
| --- |
| [Architecture Overview i](#_Toc335237077)  [1 Introduction 4](#_Toc335237078)  [**1.1** **Overview** 4](#_Toc335237079)  [**1.2** **Mission** 4](#_Toc335237080)  [2 Design 4](#_Toc335237081)  [**2.1** **Overview** 4](#_Toc335237082)  [**2.2** **Composite Keys** 4](#_Toc335237083)  [2.2.1 Composite Row Keys 4](#_Toc335237084)  [2.2.2 Composite Column Keys 4](#_Toc335237085)  [2.2.3 Byte Array Conversion 5](#_Toc335237086)  [2.2.4 Data Graph Assembly 5](#_Toc335237087)  [2.2.5 Data Graph State 5](#_Toc335237088)  [2.2.6 Row Key Generation 5](#_Toc335237089)  [2.2.7 Graph Slice Query 5](#_Toc335237090)  [**2.3** **Diagrams** 5](#_Toc335237091) |

| Table of Figures |
| --- |
| **No table of figures entries found.** |

# **Introduction**

## **Overview**

Distributed “cloud” databases allow for a new level of scalability and flexibility. These new sparse, columnar databases support a practically unlimited number of columns within a single table, and managing hundreds or even thousands of column name/qualifiers within a client application can be a significant challenge.

CloudGraph™ provides services and infrastructure to impose the structure of your business domain model, regardless of its complexity, as a service layer over various supported big-table style “cloud” databases, such as HBase and Cassandra. Application developer’s deal with higher-level typed structures with meaning within the application domain, rather than row and column qualifiers and values, typically manipulated as un-typed Java byte arrays.

Numerous best practices have evolved out of the HBase open-source software ecosystem several targeting specific strengths of these data stores and some accommodating various weaknesses, such as limited support for ACID transactions e.g. only across a single HBase row. Other critical best practices involve the use of column families and in particular the use and format of composite row and column keys. Row key design in particular involves critical decisions affecting the current and future query capabilities of a table and in general the performance and even distribution of table data across regions in a cluster.

The CloudGraph™ implementation encapsulates many HBase best practices in each of these areas and provides a framework within which to encapsulate future best practices as they evolve. Complexities of terse and efficient physical row and column key generation are completely hidden and the client user is provided with a meaningful, standards-based API generated from one or more domain-specific business models.

.

## **Mission**

Relational database design practices have long taught us to subdivide our business domains into meaningful entities and to add attributes that describe each entity within the business context. The need for meaningful business entities exists regardless of the capabilities or structure of a particular data store.

Imagine taking an average sized relational database composed of 30-40 tables and 200-300 columns and compressing this into a single tabular structure such as a spreadsheet. This is the type of challenge we face as we leverage the new sparse, columnar, distributed or “cloud” databases such as Apache HBase and Cassandra.

# **Design**

## **Composite Keys**

Several critical best practices involve the use and format of composite row and column keys. Row key design in particular involves critical decisions affecting the current and future query capabilities of a table.

### Key Generation

The initial creation and subsequent reconstitution for query retrieval purposes of both row and column keys in CloudGraph™ is efficient, as it leverages byte-array level API in both Java and the current underlying SDO 2.1 implementation, [PlasmaSDO™](http://plasma-sdo.org). Both composite row and column keys are composed in part of structural metadata, and the lightweight metadata API within [PlasmaSDO™](http://plasma-sdo.org) contains cached lookup of all basic metadata elements including logical and physical type and property names as Java byte arrays.

### Composite Row Keys

TBD

### Composite Column Keys

In order to persist any arbitrary Data Graph of any complexity in a single HBase row, every column name must be "overridden" to uniquely identify each field. This is because every column key in a row must be unique in HBase, as a row is essentially a qualifier/value pair map. Not only must the column keys for fields within a single entity be unique, but every column key must be unique for every field where many instances of the same entity type exist within the same data graph. For example Profile->Person->Contact->Address where Profile is the root entity of the graph and a Person has multiple contacts (home, business, etc...), and every field of every Contact entity is mapped to an HBase column and therefore must be unique.

Though each Data Object has a <a href="http://docs.oracle.com/javase/6/docs/api/java/util/UUID.html" target="#">UUID</a>, a unique numeric sequence ID is used as being far more efficient in terms of length, and also allowing multiple rows to "line-up" in a columnar fashion within an HBase table. This organization is helpful when viewing an HBase table columns in a relational mapping of spreadsheet based tool for debugging or analysis.

## **Complex Filters**

### Row Predicate Filters

### Graph Slice Filters

TBD

## **Data Graphs**

### Data Graph Assembly

CloudGraph™ assembles complex graph structures from low level cloud database column qualifier-value pairs according to a given business domain model.

### Data Graph State

TBD

### Data Conversion

CloudGraph™ handles the conversion and formatting of byte arrays, required by most cloud database API’s, to and from standard Java primitive and other data types specified in your business model. CloudGraph formats dates, timestamps as well as primitive types for minimal storage footprint within the target data store.

Table - Example Enterprise Java™ Project Software Layers

| Software Layer |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

.