 **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](#_Toc338179547)  [1 Introduction 4](#_Toc338179548)  [**1.1** **Overview** 4](#_Toc338179549)  [**1.2** **Mission** 4](#_Toc338179550)  [2 Design 4](#_Toc338179551)  [**2.1** **Composite Keys** 4](#_Toc338179552)  [2.1.1 Logical and Physical Names 4](#_Toc338179553)  [2.1.2 Composite Column Keys 5](#_Toc338179554)  [2.1.2.1 UUID to Sequence ID Mappings 5](#_Toc338179555)  [2.1.2.2 Column Key Fields and Sections 5](#_Toc338179556)  [2.1.2.3 Column Key Format Configuration 5](#_Toc338179557)  [2.1.3 Composite Row Keys 5](#_Toc338179558)  [2.1.4 User Defined Row Key Fields 6](#_Toc338179559)  [2.1.4.1 SDO XPath Expressions 6](#_Toc338179560)  [2.1.4.2 Row Key Format Configuration 6](#_Toc338179561)  [2.1.5 Composite Key Configuration 6](#_Toc338179562)  [2.1.5.1 Key Field Hashing 6](#_Toc338179563)  [2.1.5.2 Key Field Sequence Mapping 6](#_Toc338179564)  [2.1.5.3 Key Field Padding 6](#_Toc338179565)  [2.1.5.4 Key Field Formatting 6](#_Toc338179566)  [2.1.5.5 Key Field Data Type Conversion 6](#_Toc338179567)  [**2.2** **Partial Row-Key Scans** 6](#_Toc338179568)  [**2.3** **Filters** 6](#_Toc338179569)  [2.3.1 Row Filters 6](#_Toc338179570)  [2.3.1.1 Row Predicate Filters 6](#_Toc338179571)  [2.3.2 Column Filters 6](#_Toc338179572)  [2.3.2.1 Graph Slice Column Filters 6](#_Toc338179573)  [**2.4** **Data Graphs** 7](#_Toc338179574)  [2.4.1 Data Graph Assembly 7](#_Toc338179575)  [2.4.1.1 Data Graph Slice Assembly 7](#_Toc338179576)  [2.4.1.2 Data Conversion 7](#_Toc338179577)  [2.4.2 Data Graph State 7](#_Toc338179578)  [**2.5** **Federation** 7](#_Toc338179579)  [2.5.1 Federated Data Graphs 7](#_Toc338179580)  [2.5.1.1 External Keys 7](#_Toc338179581)  [2.5.1.2 Federated Graph Assembly 7](#_Toc338179582)  [2.5.1.3 Federated Graph Creation 8](#_Toc338179583)  [2.5.1.4 Federated Graph Modification 8](#_Toc338179584)  [2.5.1.5 Federated Graph Delete 8](#_Toc338179585) |

| 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, but 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 un-interpreted 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.

[TBD]

# **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 as well as query efficiency of a table.

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.

### Logical and Physical Names

Both composite row and column keys use physical rather than logical names for all key fields representing structural metadata within a domain model. The logical to physical name mapping supports the use of terse physical names, essential for preserving space within columnar data stores, while at the same time preserving the often more intuitive logical names within a domain model. Physical name and other logical name mappings are facilitated within [PlasmaSDO™](http://plasma-sdo.org) through a UML profile [Alias](http://plasma-sdo.org/org/plasma/sdo/Alias.html) stereotype used to annotate several model elements.

### Composite Column Keys

Apart from row cell versioning and various other powerful underlying features, each row in a sparse columnar data store could be simply thought of as a name/value pair map. And because every column key in a row must be unique, in order to persist any Data Graph of arbitrary complexity in a single row, the column qualifiers or keys must be "overridden" to uniquely identify each attribute within a data model. For example, not only must the column keys for attributes within a single model entity be unique, but every column key must be unique for every model attribute 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. [TBD create a diagram here]

#### UUID to Sequence ID Mappings

In order to save storage space and reduce complexity, though each Data Object has a [UUID](http://docs.oracle.com/javase/6/docs/api/java/util/UUID.html), rather than using these within composite column keys, UUIDs are mapped to generated integral sequence ID numbers unique to a Type. The sequence numbers are then used within the column-key data section. Sequence numbers are far more efficient that UUID’s in terms of length, and also allow multiple rows to "line-up" in a columnar fashion when displaying raw tabular data. Several tools exist which map cloud data base tables to spreadsheet or relational style displays useful for debugging and analysis. See [Toad For Cloud](http://toadforcloud.com).

#### Column Key Fields and Sections

A column key section is a segment or area with a composite column key composed of composite column key fields. Every composite column key has both a metadata section and a data section. The metadata section contains fields which *describe* and identify a specific attribute within a domain model. And the data section contains fields which identify a particular *instance* of an attribute within a domain model.

##### Column Key Metadata Section

The metadata section uses physical names over logical names. And domain model physical names represent a logical to physical mapping which supports the use of terse physical names, essential for preserving space within columnar data stores. The following pre-defined fields are part of the composite column key metadata section.

* URI - the model namespace resource identifier
* Type Name – entity type physical name, unique within a namespace
* Property Name – the attribute physical name, unique within a Type

##### Column Key Data Section

The following pre-defined fields are part of the composite column key data section.

* Sequence ID – the (instance) Data Object integral sequence number.

Note that sequence numbers are typically appended to the column key to enable the use of various column qualifier “prefix” filters such as those found in HBase.

#### Column Key Format Configuration

Most composite column key fields can be configured using various non-cryptographic hashing, whitespace and numeric padding and other formatting related settings. See [Composite Key Configuration](#_Composite_Key_Configuration).

### Composite Row Keys

The design of composite row keys involves critical decisions affecting the current and future query capabilities as well as query efficiency of a table. In addition to several pre-defined fields, The CloudGraph™ implementation supports user defined row key fields which specify the composite row-key generation characteristics for a specific Data Graph type within a table, adding another query able field or "dimension" to the key.

### User Defined Row Key Fields

Each user defined field maps a “path” within each Data Graph instance to a data property within the Data Graph. The property path is an SDO XPath expression which may contain any number of predicates with various supported relational and other logic. At runtime the field configuration is interrogated and the property path is resolved to a particular data value within the graph instance and set into the configured position within the composite row key, effectively adding another "dimension" to the key. For HBase, this flexible user defined composite row-key approach helps facilitate the important HBase partial key-scan functionality which greatly improves query performance by avoiding HBase table scans. See [Partial Row Key Scans](#_Partial_Row-Key_Scans).

#### SDO XPath Expressions

An SDO [XPath](http://plasma-sdo.org/org/plasma/sdo/xpath/package-summary.html) property path expression identifies an SDO property within a Data Graph. The expression may traverse any number of nodes within the data graph and may contain any number of predicates with various supported relational and other logic.

#### Row Key Format Configuration

Most composite row key fields can be configured using various non-cryptographic hashing, whitespace and numeric padding and other formatting related settings. See [Composite Key Configuration](#_Composite_Key_Configuration).

### Composite Key Configuration

[TBD]

#### Key Field Hashing

[TBD]

#### Key Field Sequence Mapping

[TBD]

#### Key Field Padding

[TBD]

#### Key Field Formatting

[TBD]

#### Key Field Data Type Conversion

[TBD]

## **Partial Row-Key Scans**

## **Filters**

### Row Filters

[TBD]

#### Row Predicate Filters

[TBD]

### Column Filters

[TBD]

#### Graph Slice Column 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 Slice Assembly

#### 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.

### Data Graph State

A minimal set of "state" information is persisted with each data graph in order to reduce overall storage space. In general mappings from various space intensive properties required for graph management, such as UUIDs, are mapped to integral values and the mapping stored in specific graph management columns within each row. For more information on UUIID mappings see UUID to Sequence ID Mappings.

[TBD]

## **Federation**

Due to the emergence of various client libraries supporting multi-row ACID transactions for various cloud data stores as well as more deployment intensive approaches such as co-processor libraries for HBase, processing federated write operations across multiple cloud data store tables is now possible to a limited extent, and is evolving rapidly.

### Federated Data Graphs

The tremendous flexibility afforded by cloud data stores like HBase allow for the storage of Data Graphs of arbitrary complexity and size. Under CloudGraph™, an entire data graph may be stored within a single table row, but may also span multiple rows within a single table, or even multiple rows federated across multiple data store tables. Federation is enabled based on various configuration settings within the CloudGraph™ configuration, and therefore no special model annotations are necessary.

Federation is useful where various distinct portions of the domain model, for example shared media content, must be isolated from other parts of a model, for example a package containing user profile related entities. This is often because of widely varying usage scenarios across these respective packages. For example the media content is shared by many users and the profile package is accessible only to a single user, its owner. In this case the CloudGraph™ configuration could be federated isolating shared media content within one cloud data store table, and user profile data within another cloud data store table.

In summary based on configuration settings, a data graph may be persisted:

* Within a single table row
* Across multiple rows of a single table
* Across multiple rows and multiple tables

#### Federated Graph Creation

As a federated graph is persisted for the first time, data object nodes are ordered based on a detection algorithm which establishes the nature of the relations or associations between their respective types. For example where a source node is related directly or indirectly to a target node through only singular relation(s), for a newly created graph the target node is ordered and therefore created first. And when the target of an association resides in a data sore table external to the current table context, a new row key and context is created. The row state is then mapped and persisted within the graph management or *state* elements within the current row for later de-referencing. See 2.4.2 Data Graph State for more information on graph state.

#### Federated Graph Assembly

As a graph or graph slice is assembled in response to a query, when en external key property is detected as part of the graph selection criteria, the current assembly context is mapped and a new context is determined based on 1.) the target or opposite property and 2.) the CloudGraph™ configuration containing the target entity URI and type name. The target configuration is then used to determine the target cloud data store table, and construct a row key for the new assembly context. Assembly then proceeds within the context of the new row until 1.) Another external key is detected, 2.) The current branch is complete and the next branch is mapped to a different context or 3.) The boundary of the graph is reached.

For configurations supporting federated graph assembly, the composite row key must contain the predefined UUID row key field.

#### Federated Graph Modification

[TBD]

#### Federated Graph Delete

[TBD]

Table - Example

| COLUMN |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

.