Large-Scale and Multi-Structured Databases

Column Databases

Design Tips

Prof. Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

IN SUPREMÆ DIGNITATIS 1343

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Column DB Design

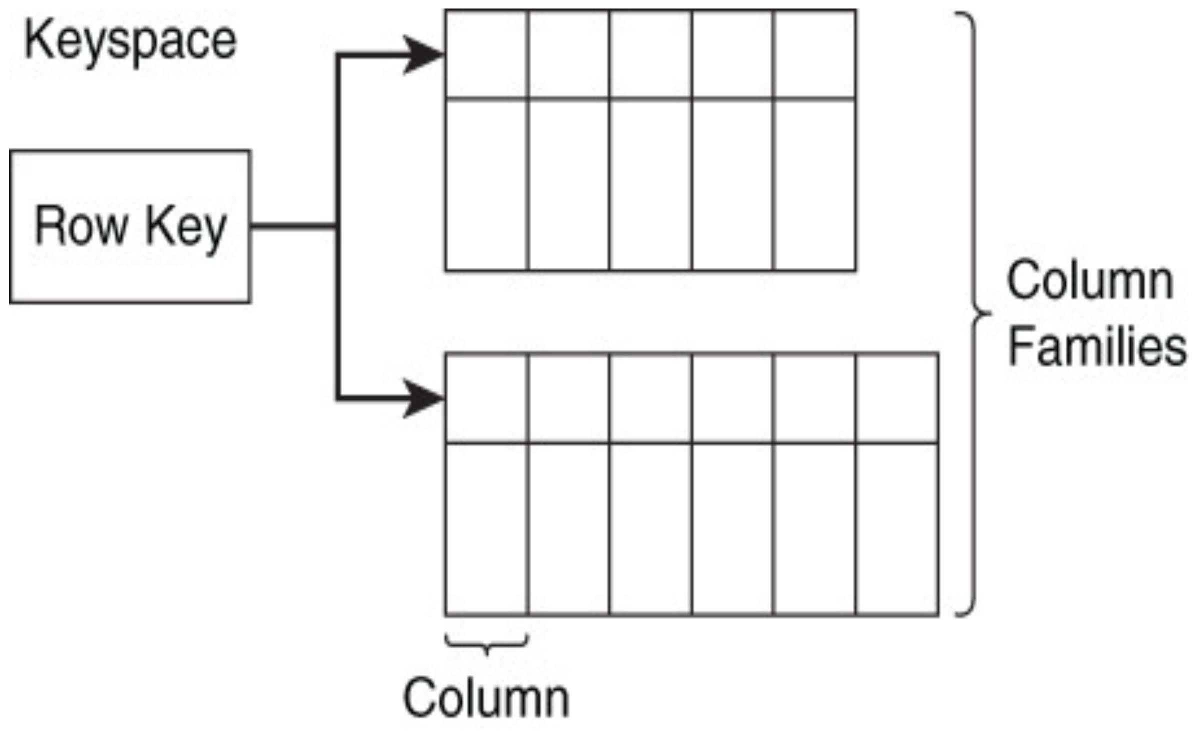


Image extracted from: “Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015

Users (and queries) Guide the DB Design

In the following, we show some examples of ***typical queries*** that may led to choose ***column DB***:

- How many new orders were placed in the Northwest region yesterday?
- When did a particular customer last place an order?
- What orders are en route to customers in London, England?
- What products in the Ohio warehouse have fewer than the stock keeping minimum number of items?

Users (and queries) Guide the DB Design

In the following, we show some examples of ***typical queries*** that may led to choose ***column DB***:

- How many new orders were placed in the Northwest region yesterday?

Data like "order date," "region," and "order status" can be stored in separate columns, allowing fast access to only the relevant columns without reading unnecessary data.

This minimizes I/O and speeds up the aggregation (like counting orders) specifically for the Northwest region on a particular date.

Users (and queries) Guide the DB Design

- When did a particular customer last place an order?

We only need data from the "customer ID" and "order date" columns.

A columnar database enables efficient access to these specific columns, making it easy to retrieve the last order date without scanning through unrelated columns, thus speeding up the query.

Users (and queries) Guide the DB Design

- What orders are en route to customers in London, England?

Only the "order status" and "customer location" columns are needed.

Columnar databases allow fast filtering on these columns, so querying orders that are "en route" to a particular location is efficient, especially with large datasets where irrelevant columns can be skipped entirely.

Users (and queries) Guide the DB Design

- What products in the Ohio warehouse have fewer than the stock keeping minimum number of items?

This query requires access to "warehouse location," "product ID," and "inventory count" columns.

A columnar database allows quick filtering on location and fast comparison of inventory levels without accessing other product attributes, making it efficient for inventory management or alerts for stock levels.

Information Needed for the DB Design

Queries provide information needed to effectively design column family databases.
The information includes:

- Entities
- Attributes of entities
- Query criteria
- Derived values

Designers start with this information and then use ***the features of column-based*** databases management systems to ***select*** the most appropriate ***implementation***.

How to Use the Extracted Information

Entities
A single row describe the instance of a single entity.
Rows are uniquely identified by row keys.

Attributes of entities
Attributes of entities are modeled using columns.

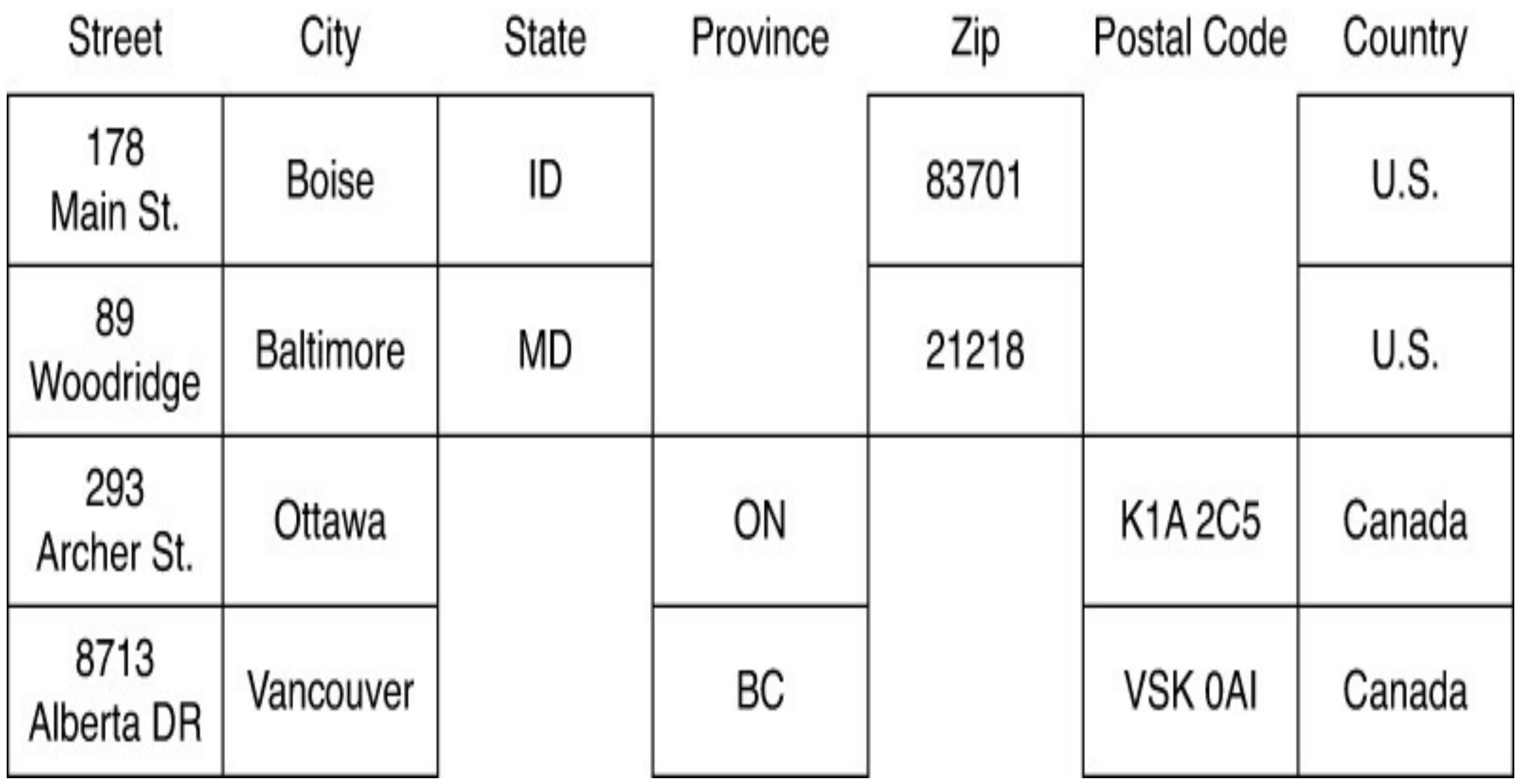
Query criteria
The selection criteria should be used to determine optimal ways to organize data with tables and column families and how to build partitions

Derived values
it is an indication that additional attributes may be needed to store derived data (example “count of orders placed yesterday by a user”).

Differences with Relational DBs

- Column databases are implemented as ***sparse*** and multidimensional ***maps***
- Columns can ***vary*** between rows
- Columns can be added ***dynamically***
- ***Joins*** are ***not*** used because ***data*** is ***denormalized***
- It is suggested to use ***separate keyspace*** for each application.

Sparse and Multi-dimensional Maps

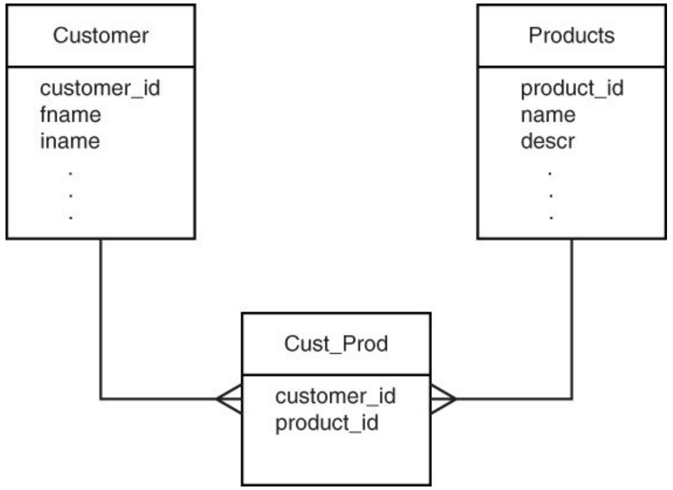


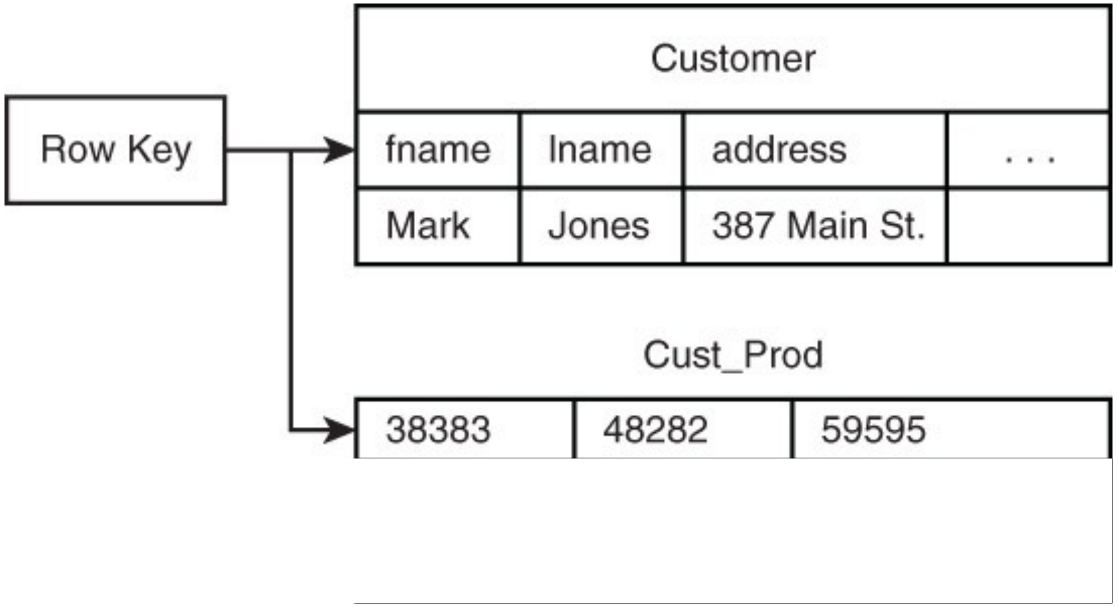
Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

Image extracted from: "Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015

Denormalization and **Valueless** Columns

Example: Many to many relationships





Relational DB

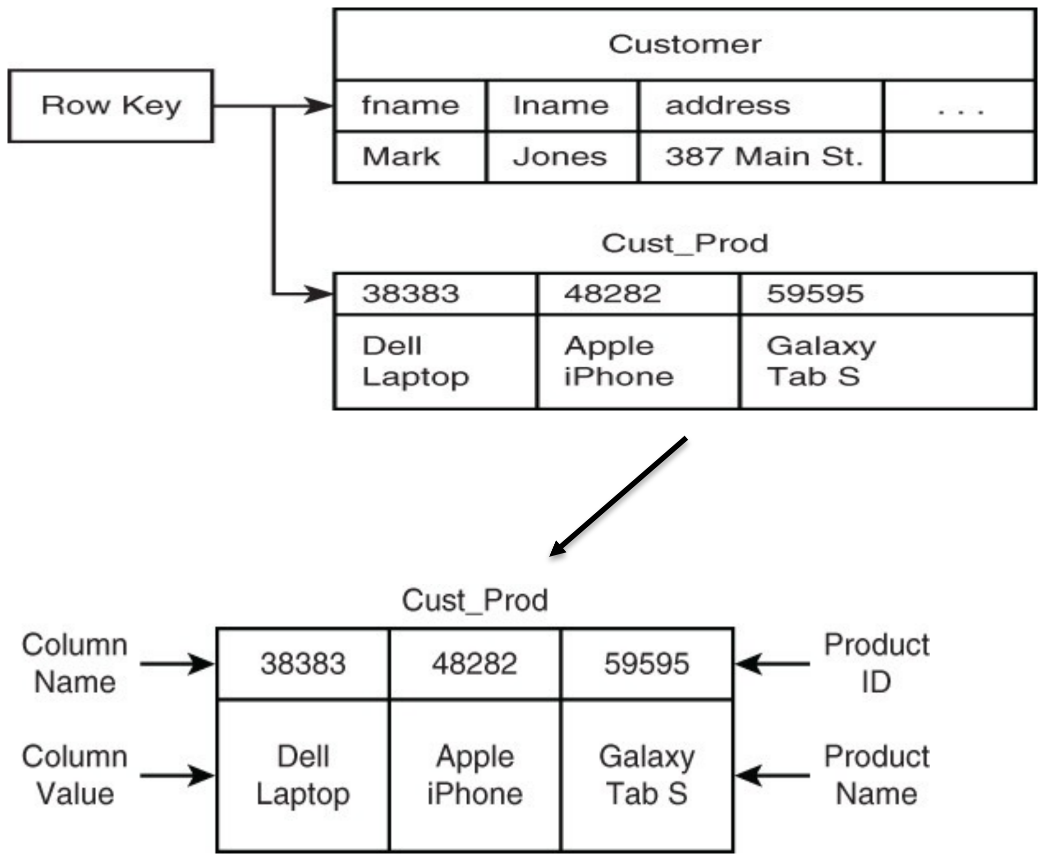
We use 3 tables, including a join table.

Column DB

In order to avoid join operations, each customer includes a ***set of column names*** that correspond to purchased products (Ids).

Use Both Column Names and Column Values to Store Data

Example: Many to many relationships

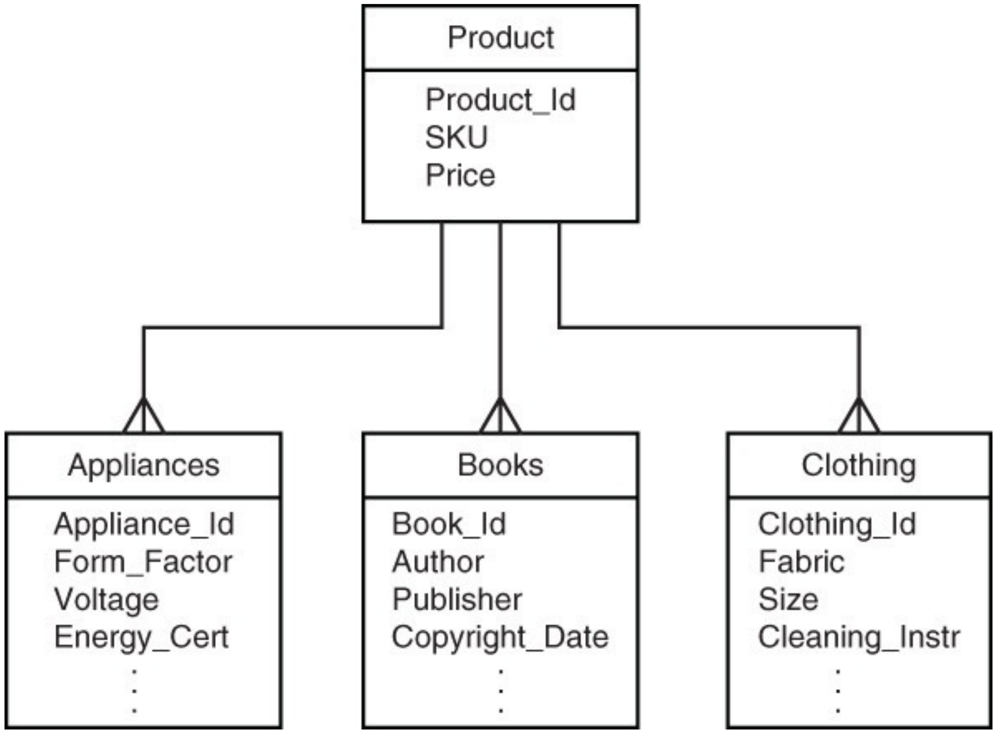


Because the column value is not used for anything else, we can store the product name there.

This can avoid the access to the product table, if we want to produce a report listing products bought by a customer .

Keeping a copy of the product name in the customer table will ***increase*** the amount of ***storage*** used (***we improve the read performance***).

Model an Entity with a Single Row



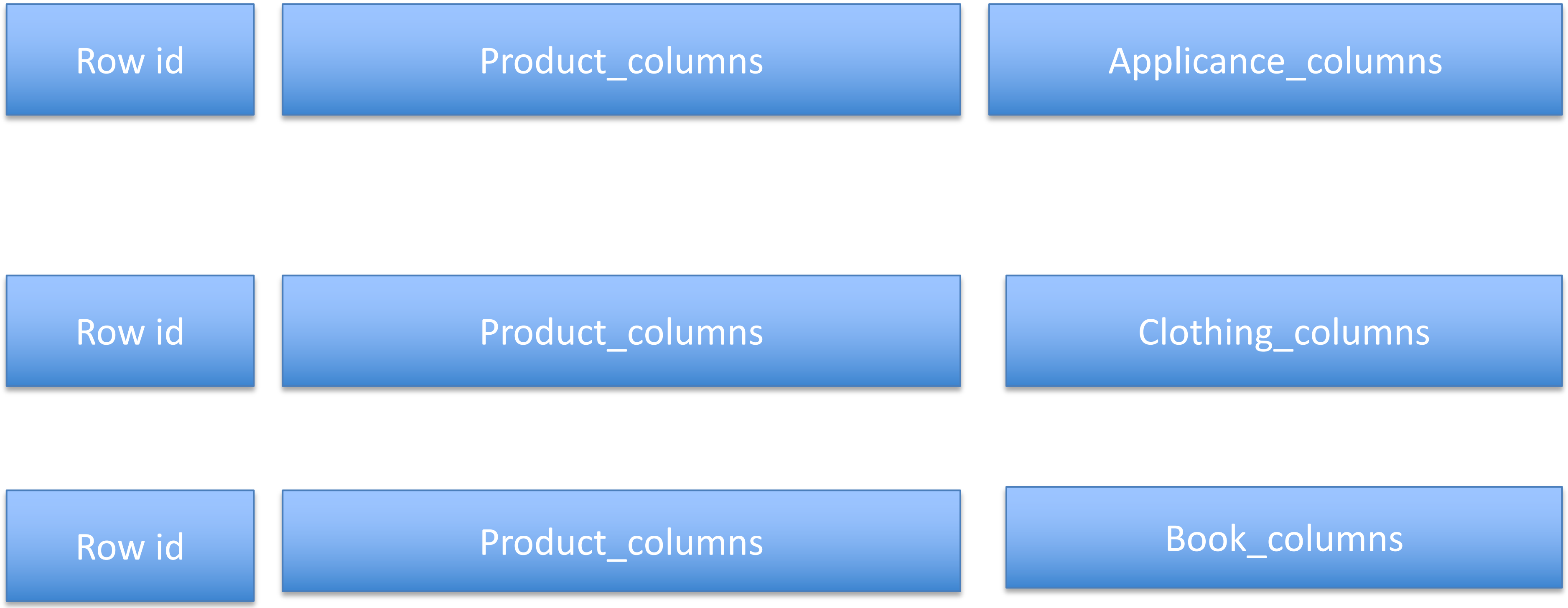
A single instance of one entity, such as a particular customer or a specific product, should have ***all its attributes*** in a ***single row***.

In this case, typical of column DBs, some rows store more column values than others.

The figure shows the classical organization in a relational DB of an entity that can assume different "***shapes***".

In order to ***ensure*** the ***atomicity*** of the operation in column DB, the same entity than in the figure must be organized in just ***one table with 4 column families***.

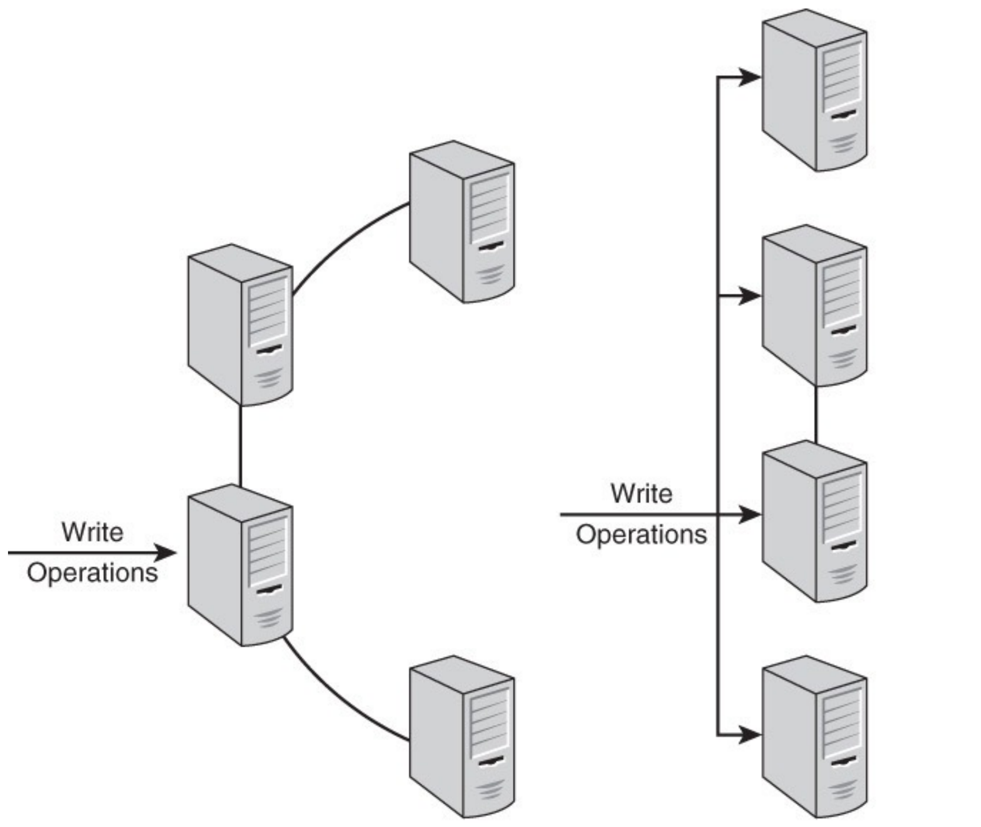
Model an Entity with a Single Row



- It is worth to notice that the ***atomicity*** is typically ensured at ***row level***!
- Strong support to ***ACID transactions*** maybe ***not supported*** by Column DBs

Avoid Hotspotting in Row Keys

Hotspotting occurs when many operations are performed on a small number of servers



(a) Hotspotting

(b) More Even Distribution

We can prevent hotspotting by ***hashing sequential values*** generated by other systems and used as row key in column DBs.

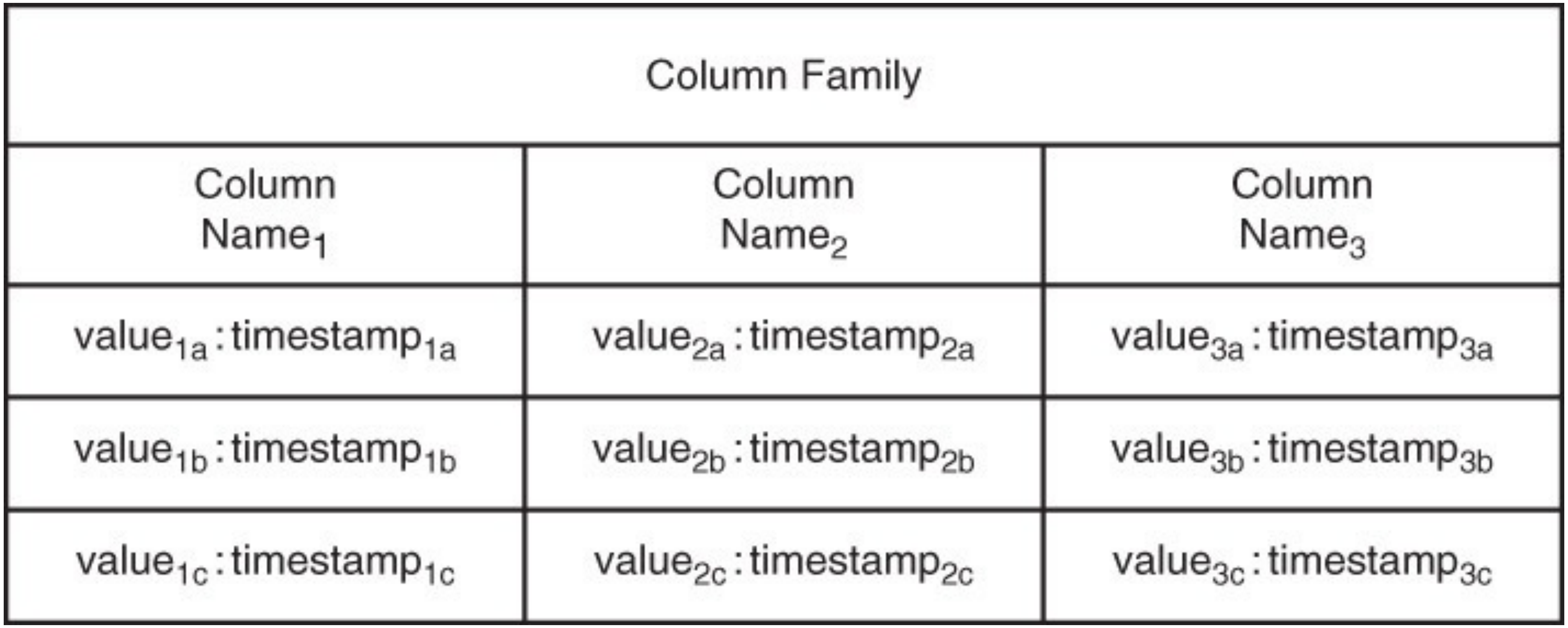
Alternatively, we could ***add a random string*** as a prefix to the sequential value.

These strategies would ***eliminate*** the effects of the ***lexicographic orde***r of the source file on the data load process.

Keep an Appropriate Number of Column Value Versions

- Some column DBMSs provides for ***column value versions***
- The number of versions maintained is controlled by ***database parameters*** (min and max number of data versions)
- The versioning parameters ***depends*** on application ***requirements***
- Versioning is useful if we need to ***roll back changes*** we did to column values.

Column Value Versions

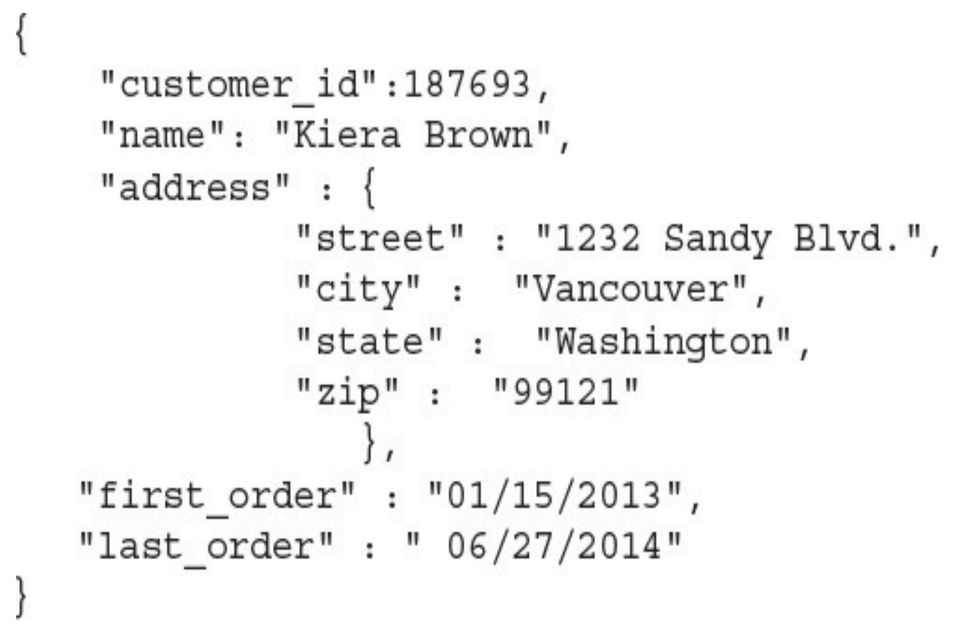


Column Family		
Column Name$_1$	Column Name$_2$	Column Name$_3$
value$_{1a}$: timestamp$_{1a}$	value$_{2a}$: timestamp$_{2a}$	value$_{3a}$: timestamp$_{3a}$
value$_{1b}$: timestamp$_{1b}$	value$_{2b}$: timestamp$_{2b}$	value$_{3b}$: timestamp$_{3b}$
value$_{1c}$: timestamp$_{1c}$	value$_{2c}$: timestamp$_{2c}$	value$_{3c}$: timestamp$_{3c}$

The ***oldest versions of values***, if needed, will be removed during ***data compaction*** procedures

Avoid Complex Data Structures in Column Values

Any kind of data structure may be stored in a column value such a JSON file:



```
{
    "customer_id":187693,
    "name": "Kiera Brown",
    "address" : {
             "street" : "1232 Sandy Blvd.",
             "city" :  "Vancouver",
             "state" :  "Washington",
             "zip" :  "99121"
               },
   "first_order" : "01/15/2013",
   "last_order" : " 06/27/2014"
}
```

Store complex structure as a column value only if you just need to store and retrieve it as is!

Notice that:

- Using ***separate columns*** for each attribute makes it easier to apply database features to the attributes (***indexing***).

- ***Separating attributes*** into individual columns allows you to use different ***column families*** if needed.

Indexing: Primary and Secondary Indexes

Index: a specific data structure in a DBMS that allows the database engine to ***retrieve*** data ***faster*** than it otherwise would.

In column databases, we can ***look up*** a ***column value*** to quickly ***find rows*** that reference that column value.

SELECT fname, lname FROM customers WHERE state = ‘OR’;

Primary indexes are indexes on the ***row keys*** of a table (created and maintained by the DBMS).

Secondary indexes are indexes created on one or more ***column value***

Either the database system or your application can create and manage secondary indexes.

Secondary Indexes Managed by the Database Management System

General and common sense rule:
"if we need secondary indexes on column values and the column family database system provides automatically managed secondary indexes, then we should use them."

Example: we can speedup this query

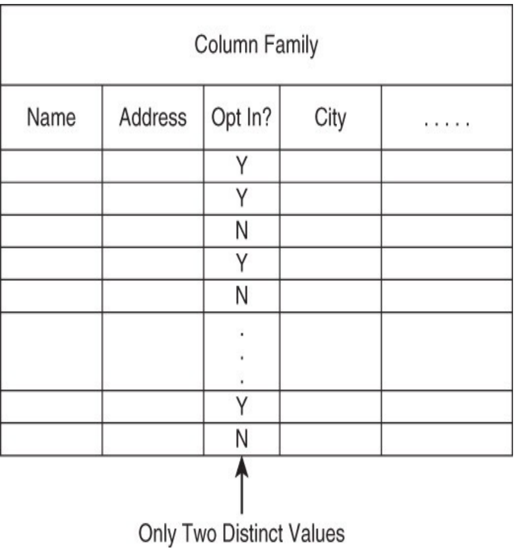
SELECT fname, lname FROM customers WHERE state = 'OR' AND lname = 'Smith'

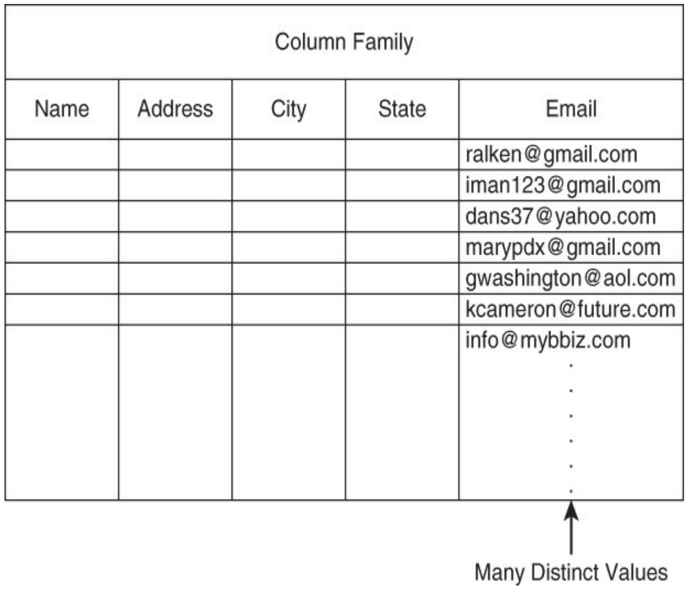
By setting an index on the *state* and on the *lname* attributes.

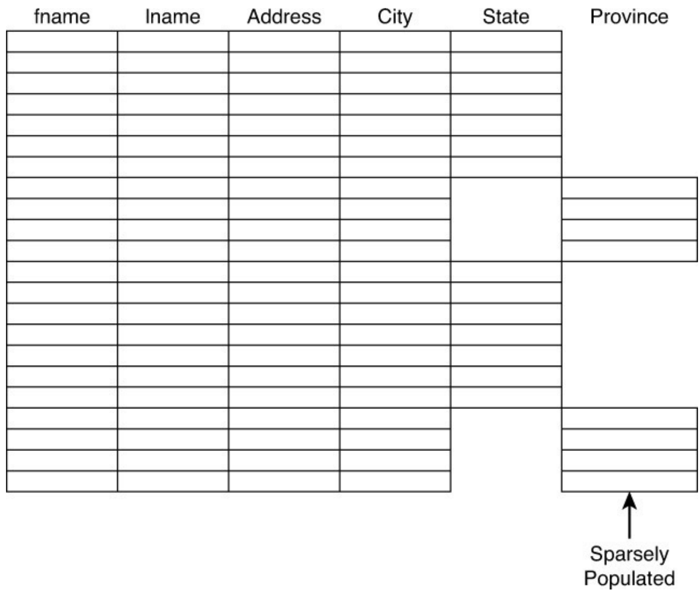
Typically, the DBMS will use the ***most selective*** index first.

The automatic use of secondary indexes has the advantage the we do not have to change our code to use the indexes, if the application requirements will change.

When avoiding to use automatically managed indexes







An index will probably not help much, especially if there are ***roughly equal numbers*** of each value.

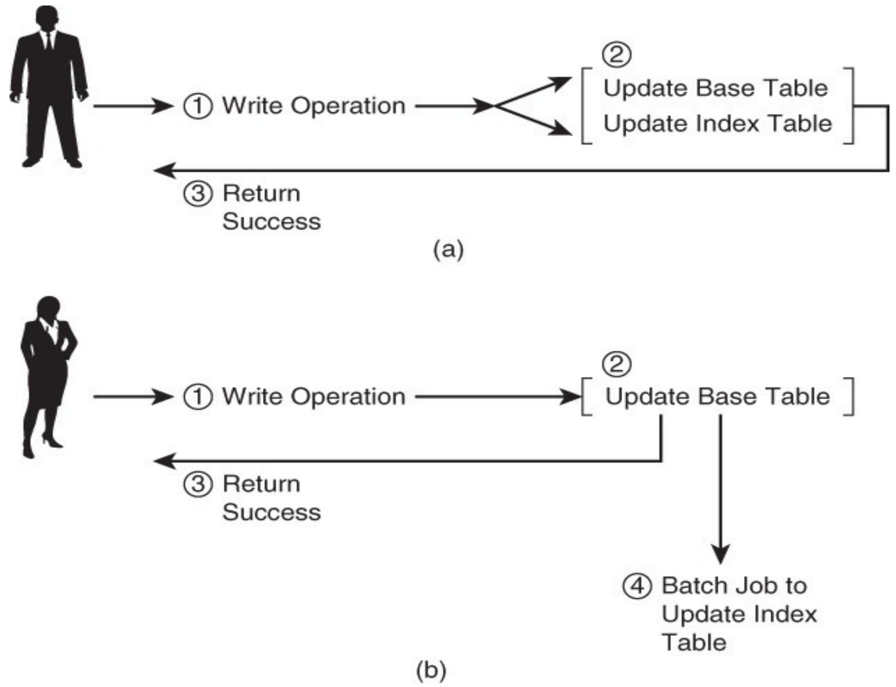
Indexes may not help much here because the index will have to ***maintain so much data*** it could take ***too much time*** to search the index and retrieve the data.

Too few values associated to a specific column. It is not worth creating and managing an index data structure for this attribute.

Create and Manage Secondary Indexes Using Tables

In this case, we have to ***explicitly create*** and manage tables to store data we would like to access via the index.

The ***programmer*** will be be ***responsible*** for maintaining the indexes and two main strategies may be adopted:



a) ***Updating an index table during write*** operations keeps data synchronized but ***increases the time*** needed to complete a write operation.

b) ***Batch updates*** introduce periods of time when the ***data is not synchronized***, but this may be acceptable in some cases.

Suggested Readings

Chapter 11 of the book "*Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015.*"

All the images used in this presentation have been extracted from the aforementioned book.