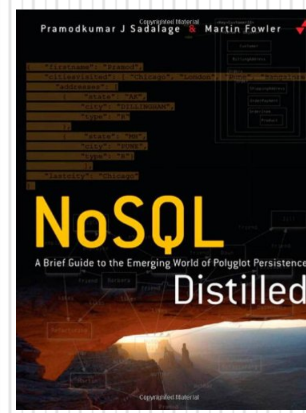
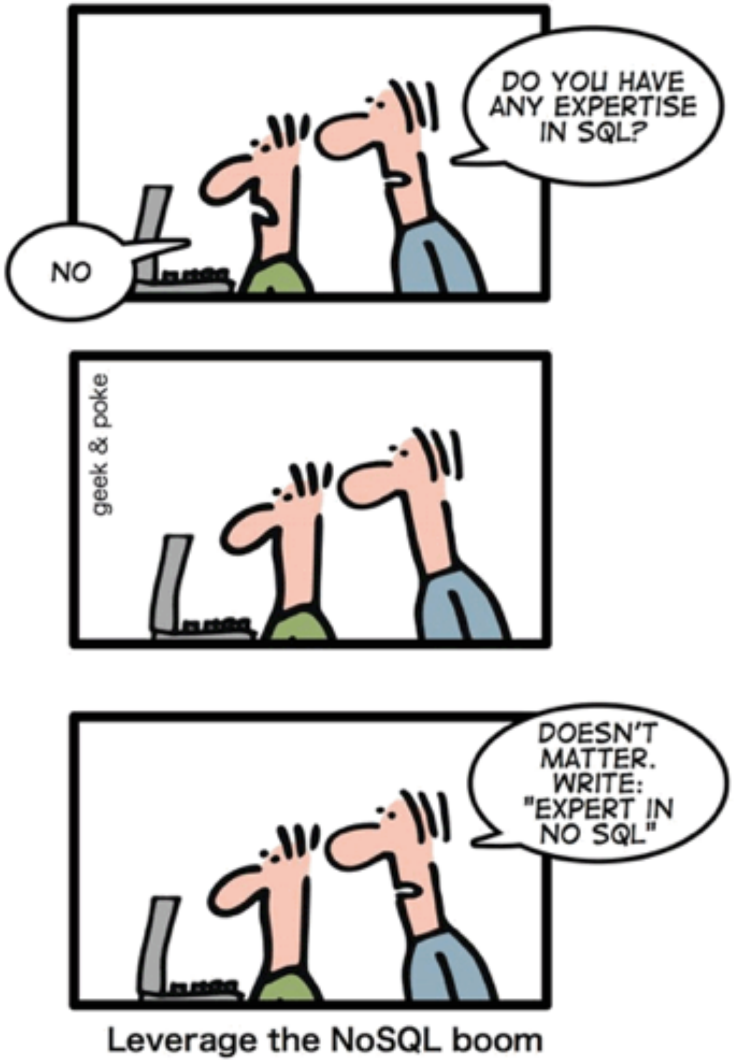
NoSQL systems: introduction and data models

Riccardo Torlone
Università Roma Tre



Leveraging the NoSQL boom

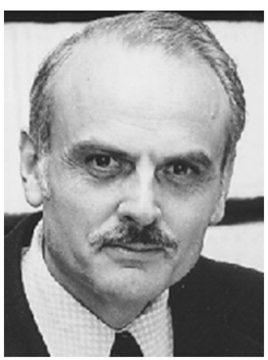


Why NoSQL?

- In the last fifty years relational databases have been the default choice for serious data storage.
- An architect starting a new project:
  - your only choice is likely to be which relational database to use.
  - often not even that, if your company has a dominant vendor.
- In the past, other proposals for database technology:
  - deductive databases in the 1980's
  - object databases in the 1990's
  - XML databases in the 2000's
  - these alternatives never got anywhere.

The Value of Relational Databases

- Effective and efficient management of persistent data
- Concurrency control
- Data integration
- A standard data model
- A standard query language



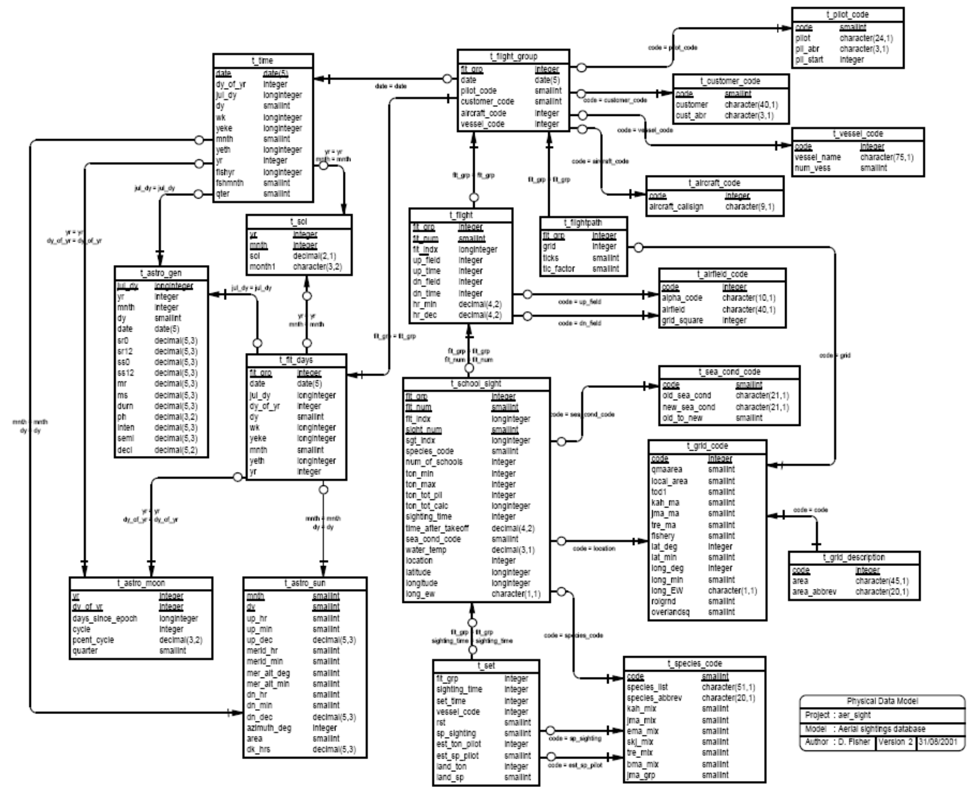
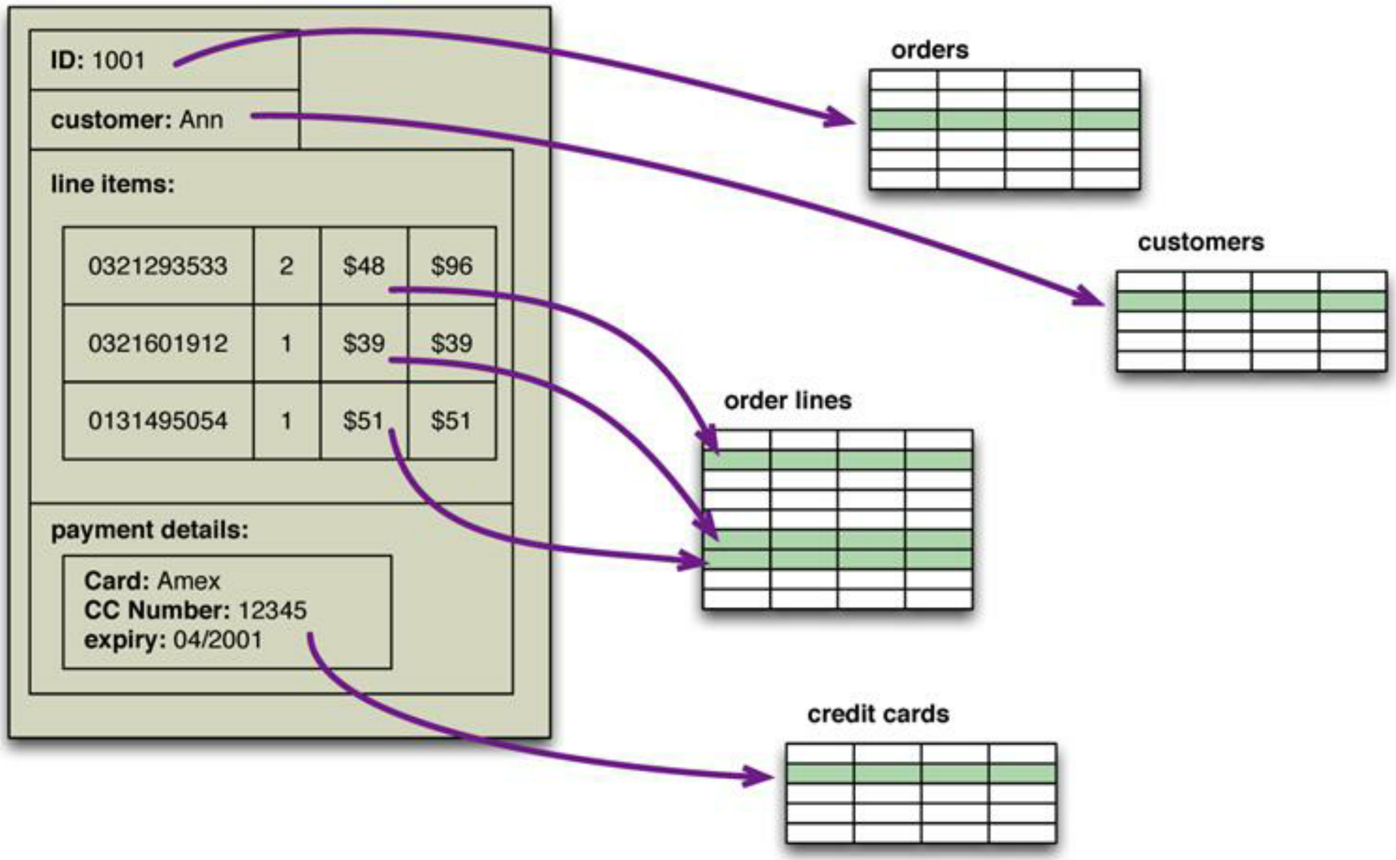


Figure 1: Entity Relationship Diagram (ERD) for the aer_sight database

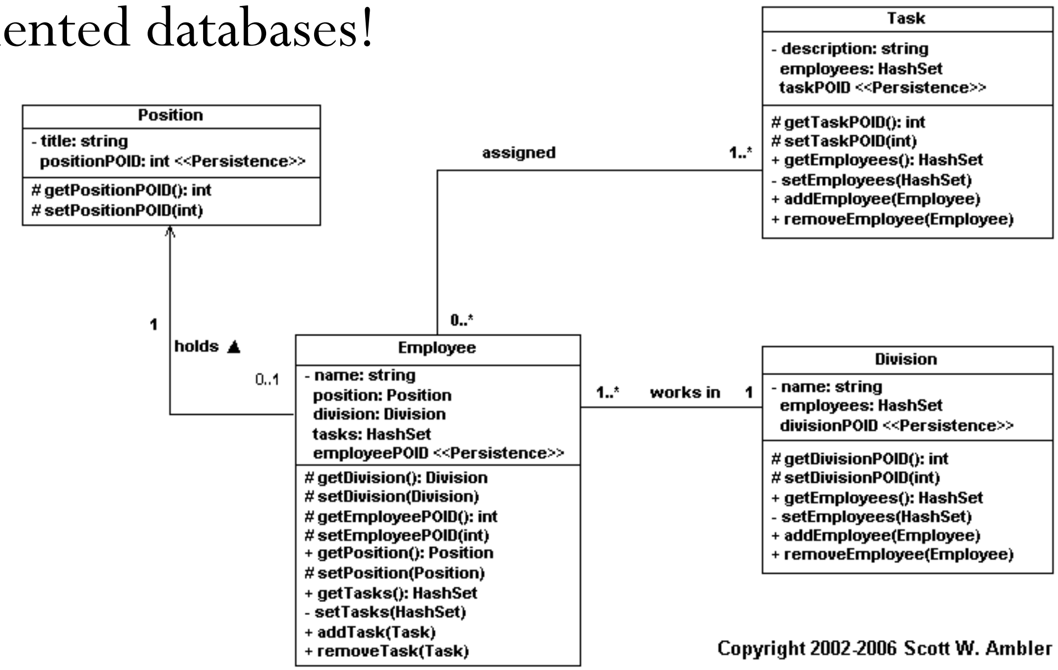
Impedance Mismatch

- Difference between the persistent data model and the in-memory data structures



A proposal to solve the problem (1990s)

- Databases that replicate the in-memory data structures to disk
- Object-oriented databases!



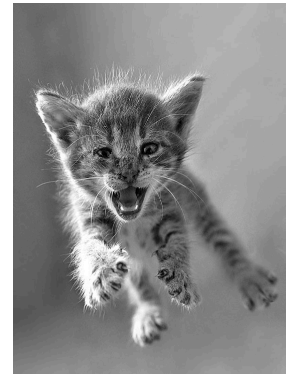
- Faded into obscurity in a few years..
- Solution emerged:
  - object-relational mapping frameworks

Evolution of applications

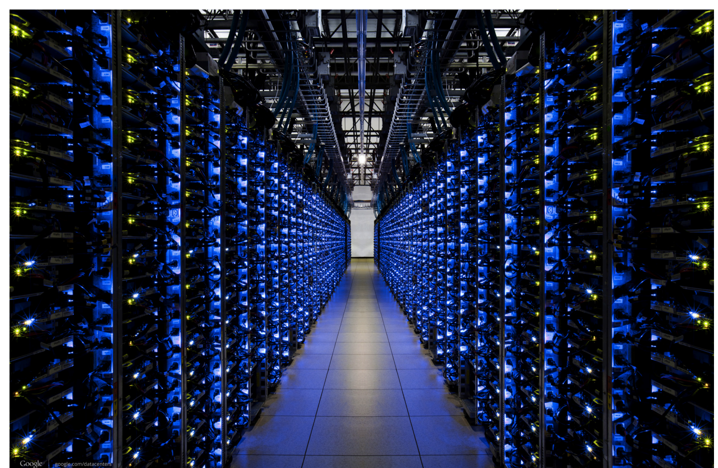
- OO databases are dead. Why?
  - SQL provides an integration mechanism between applications
  - The database acts as an integration database
  - Multiple applications one database
- 2000s: a distinct shift to application databases (SOA)
  - Web services add more flexibility for the data structure being exchanged
  - richer data structures to reduce the number of round trips
    - nested records, lists, etc.
  - usually represented in XML or JSON.
  - you get more freedom of choosing a database
    - a decoupling between your internal database and the services with which you talk to the outside world
  - Despite this freedom, however, it wasn't apparent that application databases led to a big rush to alternative data stores.

Relational databases are familiar and usually work very well
(or, at least, well enough)

Attack of the Clusters



- A shift from scale up to scale out
  - With the explosion of data volume the computer architectures based on cluster of commodity hardware emerged as the only solution
  - but relational databases are not designed to run (and do not work well) on clusters!
- The mismatch between relational databases and clusters led some organization to consider alternative solutions to data storage
- Google: BigTable
- Amazon: Dynamo



NoSQL

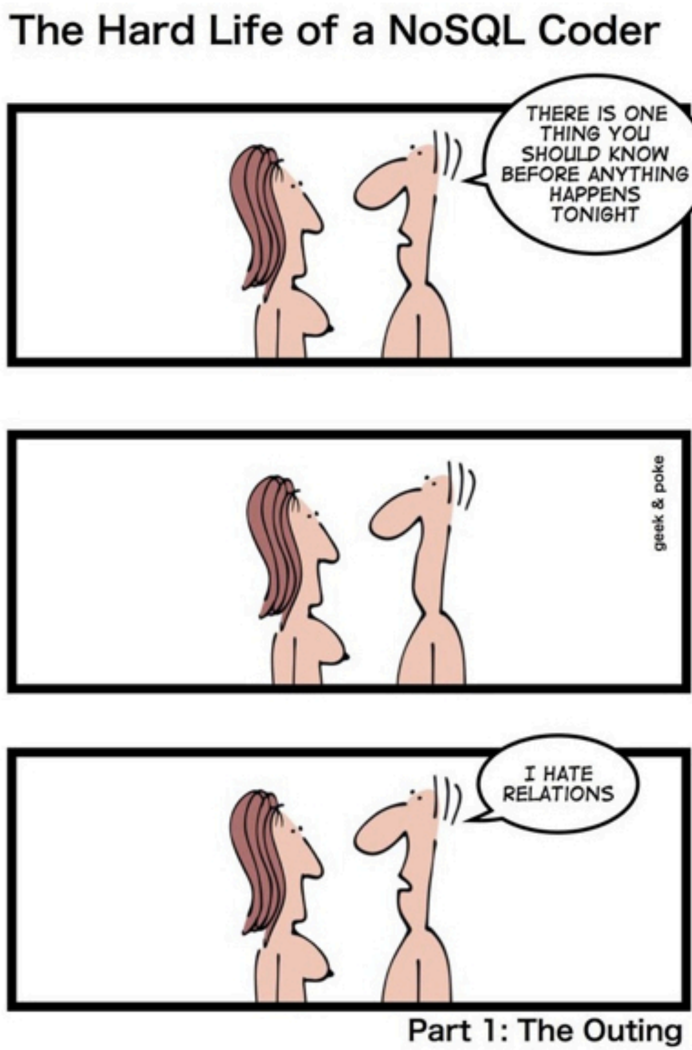
- Term appeared in the late 90s
  - open-source relational database [Strozzi NoSQL]
  - tables as ASCII files, without SQL
- Current interpretation
  - June 11, 2009: meetup in San Francisco
  - Open-source, distributed, non-relational databases
  - Hashtag chosen: #NoSQL
  - Main features:
    - Not using SQL and the relational model
    - Open-source projects (mostly)
    - Running on clusters
    - Schemaless
  - However, no accepted precise definitions
- Most people say that NoSQL means "Not Only SQL"

Key Points

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.
- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.
- The common characteristics of NoSQL databases are:
  - Not using the relational model
  - Running well on clusters
  - Open-source
  - Schemaless

[Popularity]

The non-relational world



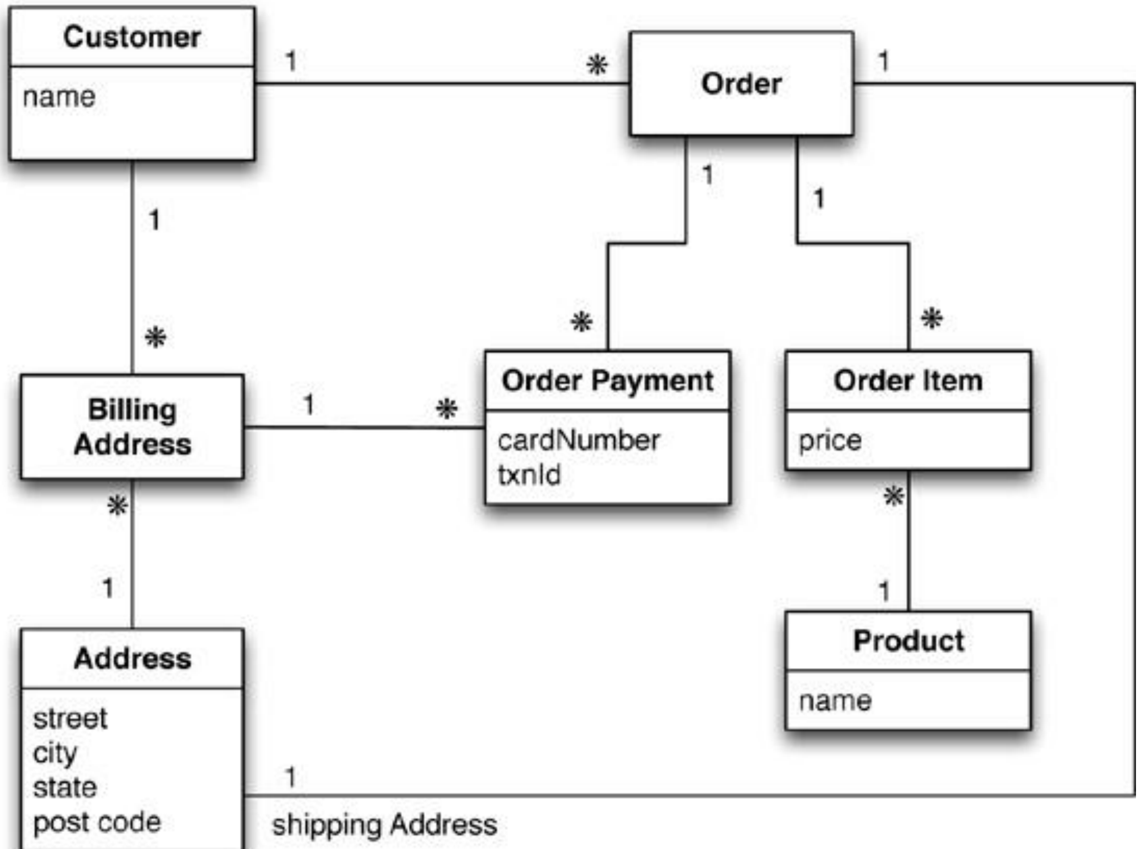
NoSQL Data Models

- A data model is a set of constructs for representing the information
  - Relational model: tables, columns and rows
- Storage model: how the DBMS stores and manipulates the data internally
- A data model is usually independent of the storage model
- Data models for NoSQL systems:
  - aggregate models
    - key-value,
    - document,
    - column-family
  - graph-based models

Aggregates

- Data as atomic units that have a complex structure
  - more structure than just a set of tuples
  - example:
    - complex record with: simple fields, arrays, records nested inside
- Aggregate in Domain-Driven Design
  - a collection of related objects that we treat as a unit
  - a unit for data manipulation and management of consistency
- Advantages of aggregates:
  - easier for application programmers to work with
  - easier for database systems to handle operating on a cluster

Example


Customer
name
Order
Order Payment
cardNumber
txnId
Order Item
price
Billing Address
Address
street
city
state
post code
Product
name
shipping Address
1
*

Relational implementation

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

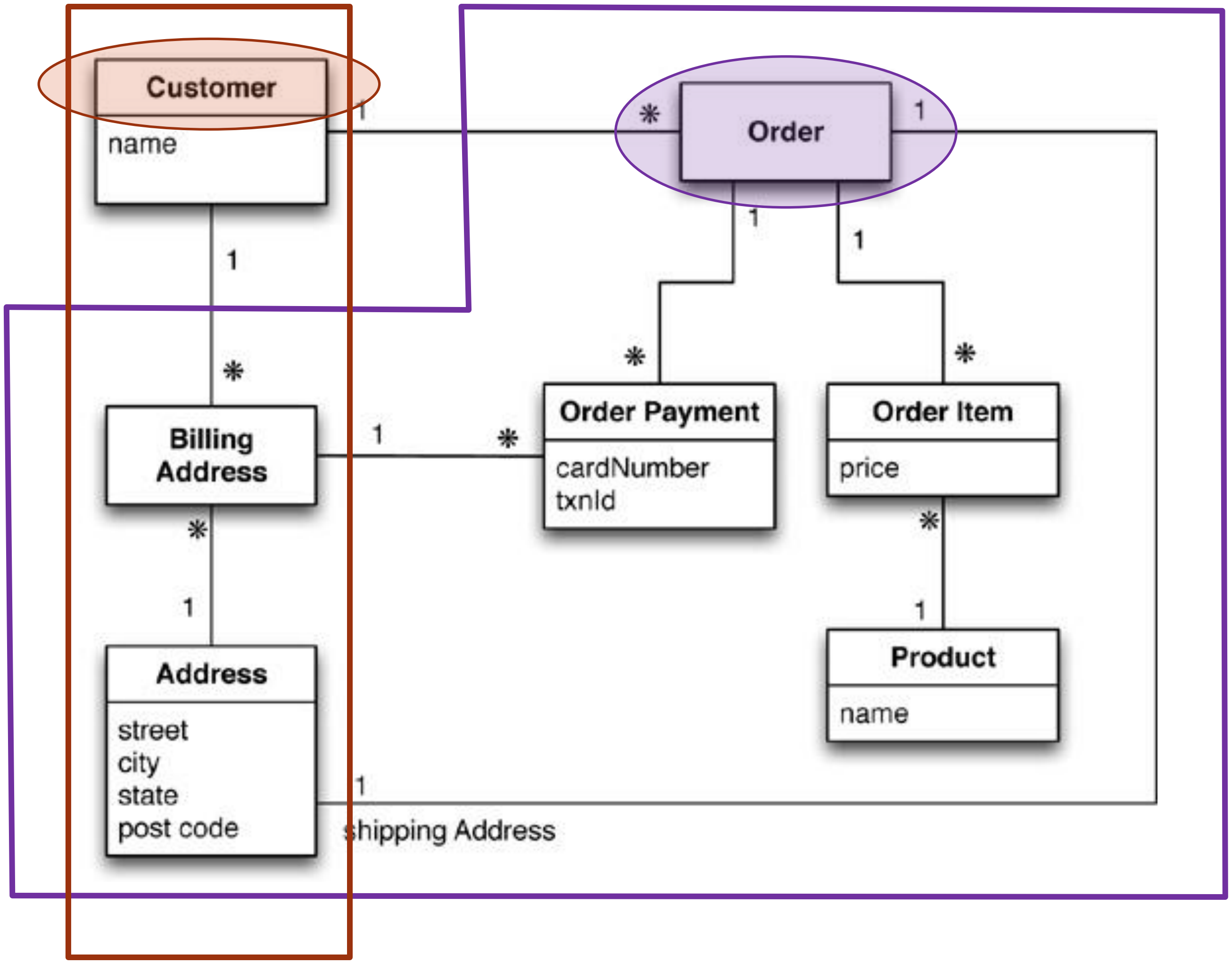
Address

Id	City
77	Chicago

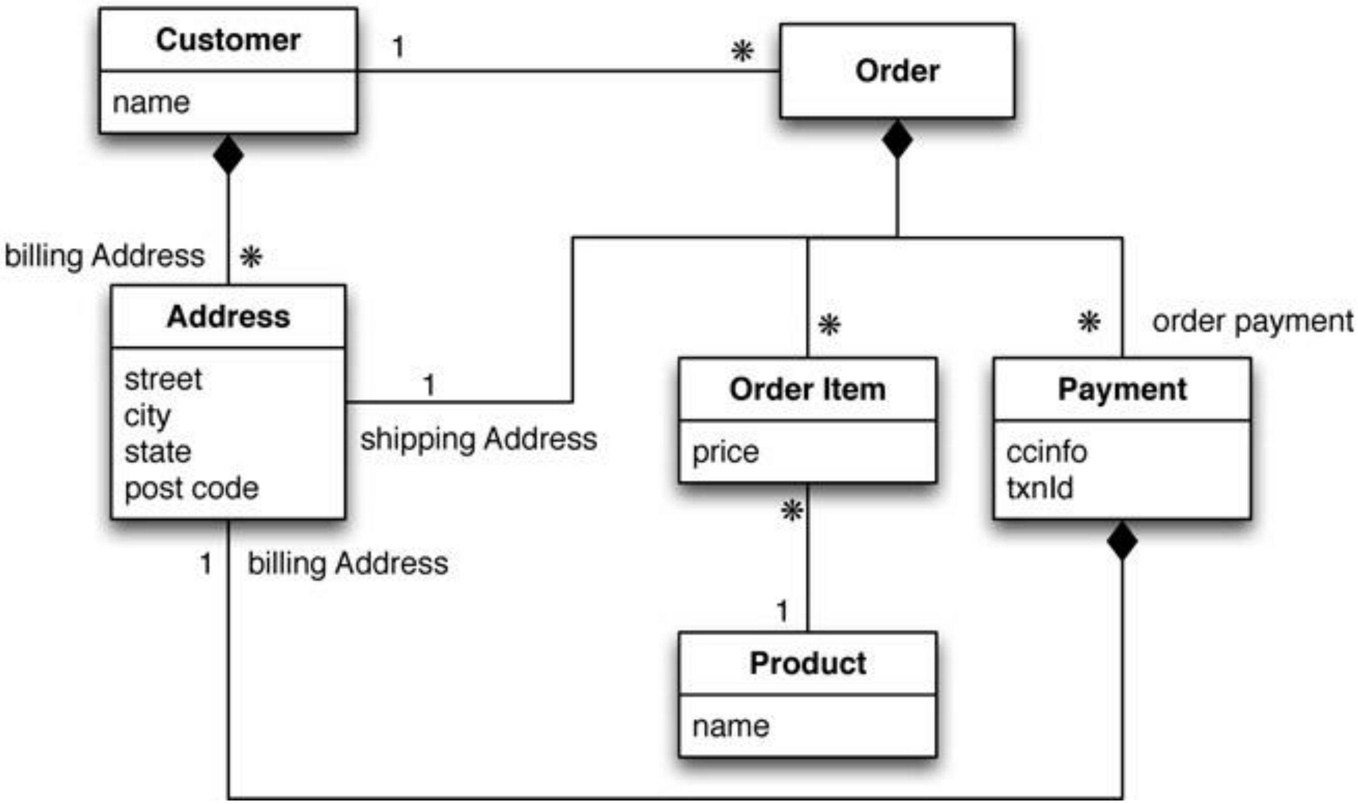
OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

A possible aggregation



Aggregate representation

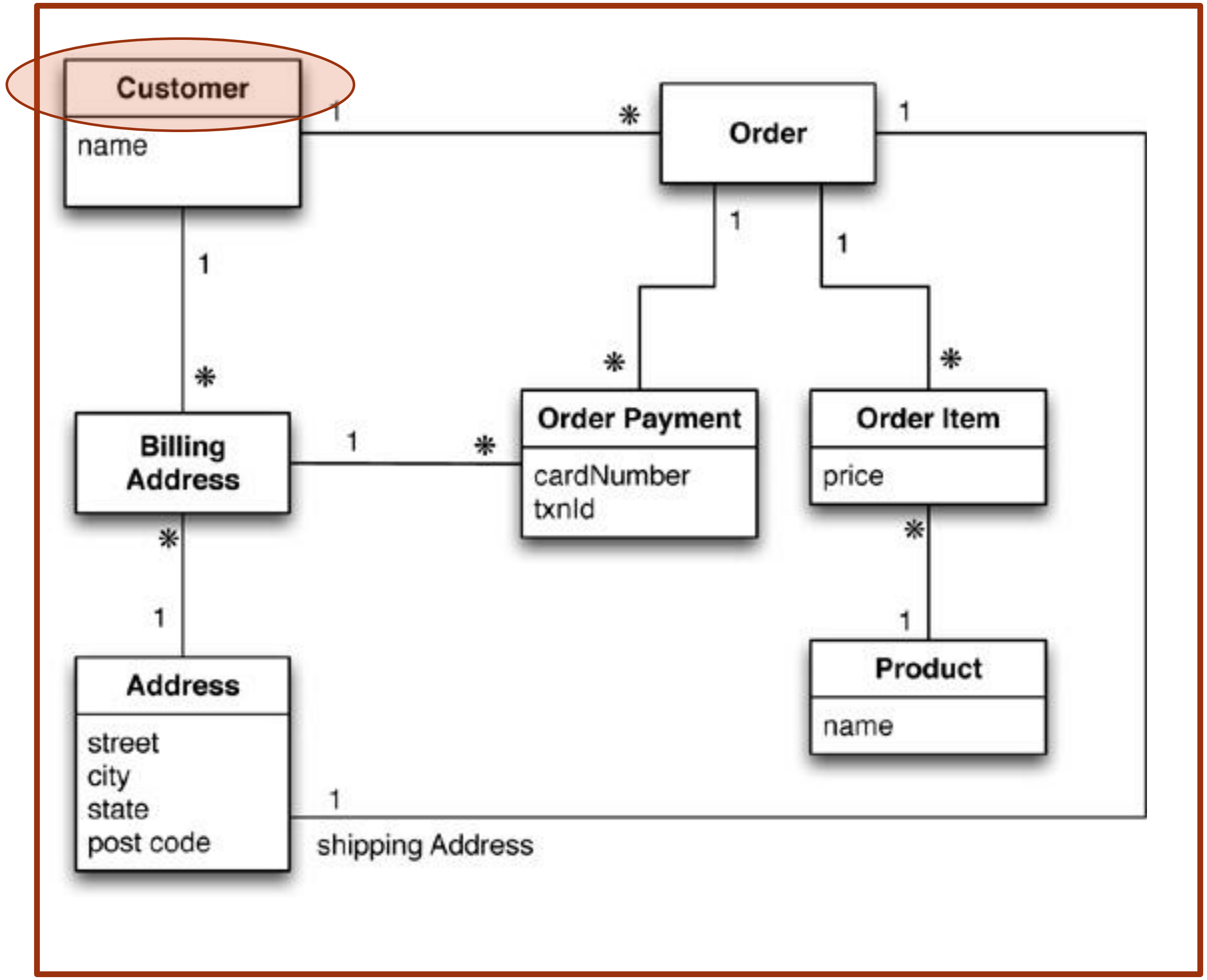


Aggregate implementation

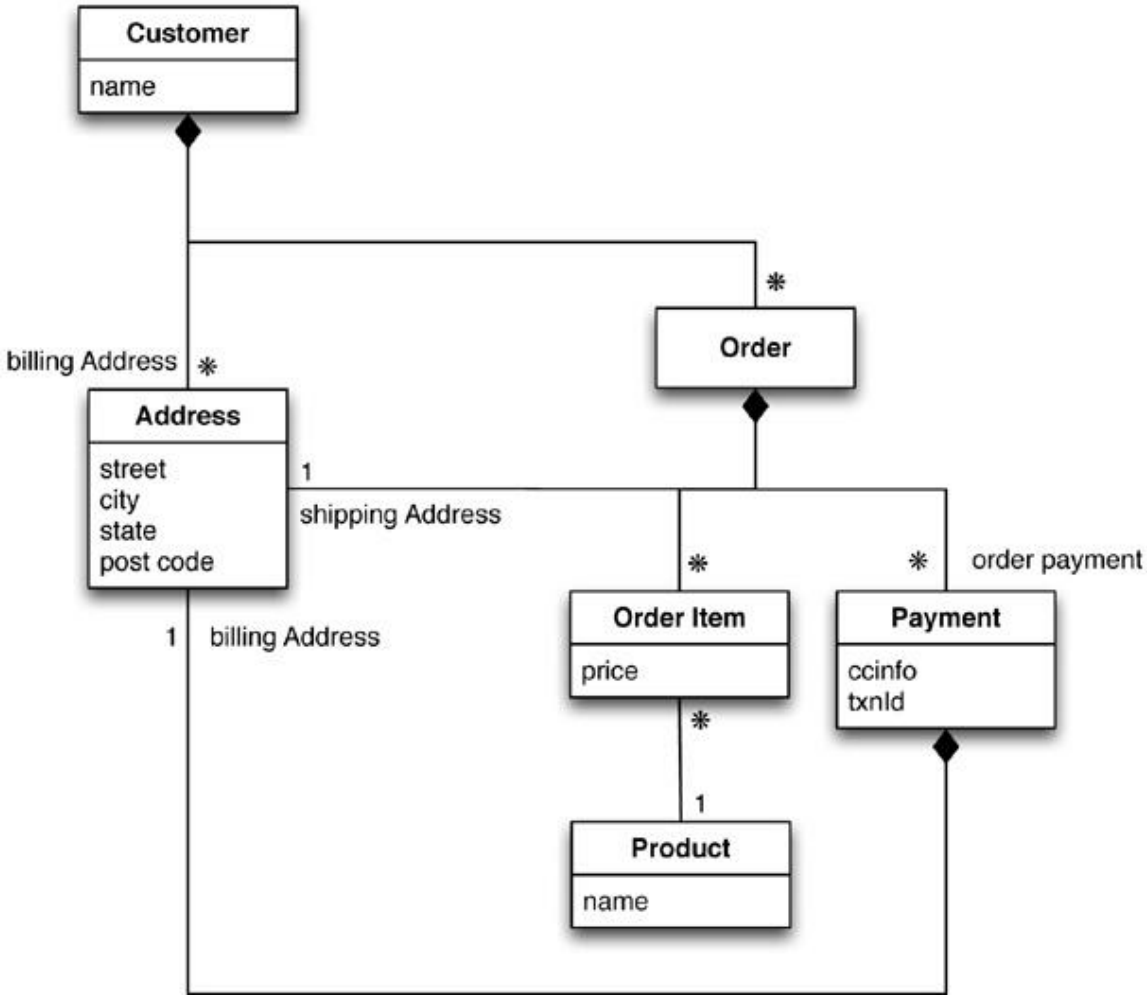
```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Another possible aggregation



Aggregate representation (2)

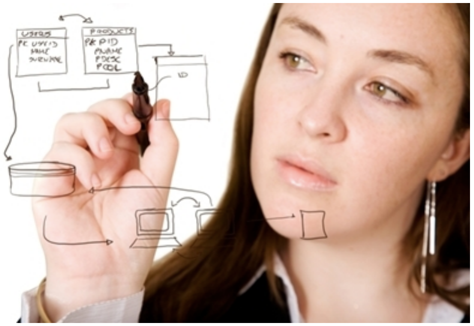


Aggregate implementation (2)

```
 // in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Design strategy

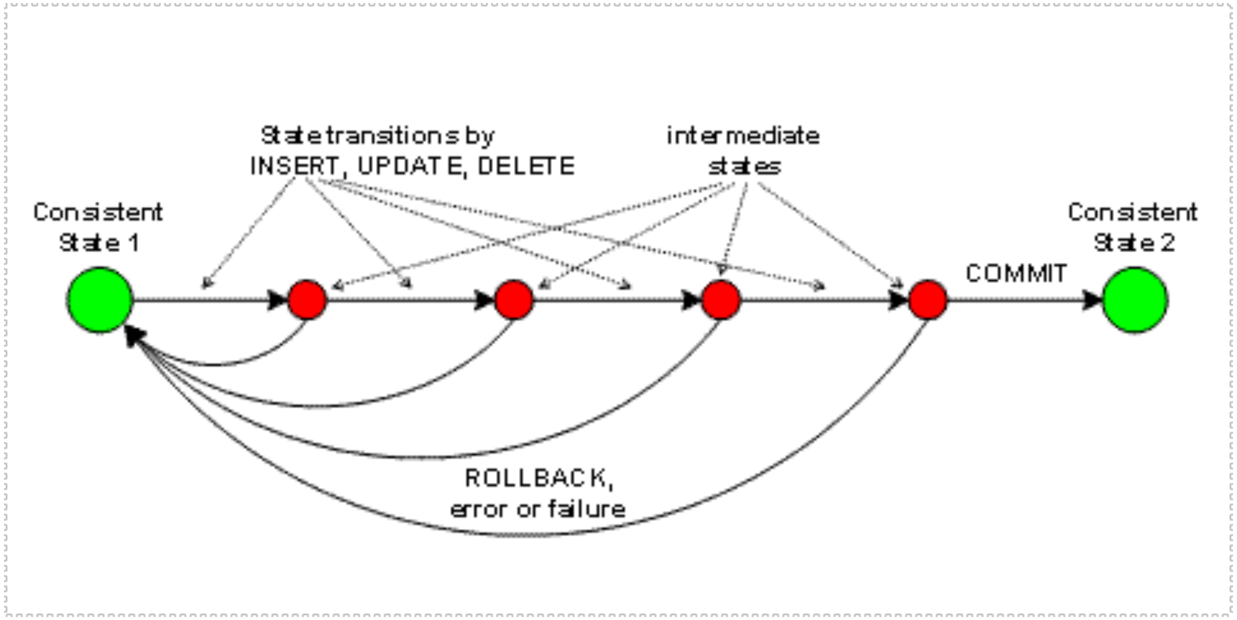
- No universal answer for how to draw aggregate boundaries
- It depends entirely on how you tend to manipulate data!
  - Accesses on a single order at a time: first solution
  - Accesses on customers with all orders: second solution



- Context-specific
  - some applications will prefer one or the other
  - even within a single system
- Focus on the unit of interaction with the data storage
- Pros:
  - it helps greatly with running on a cluster: data will be manipulated together, and thus should live on the same node!
- Cons:
  - an aggregate structure may help with some data interactions but be an obstacle for others

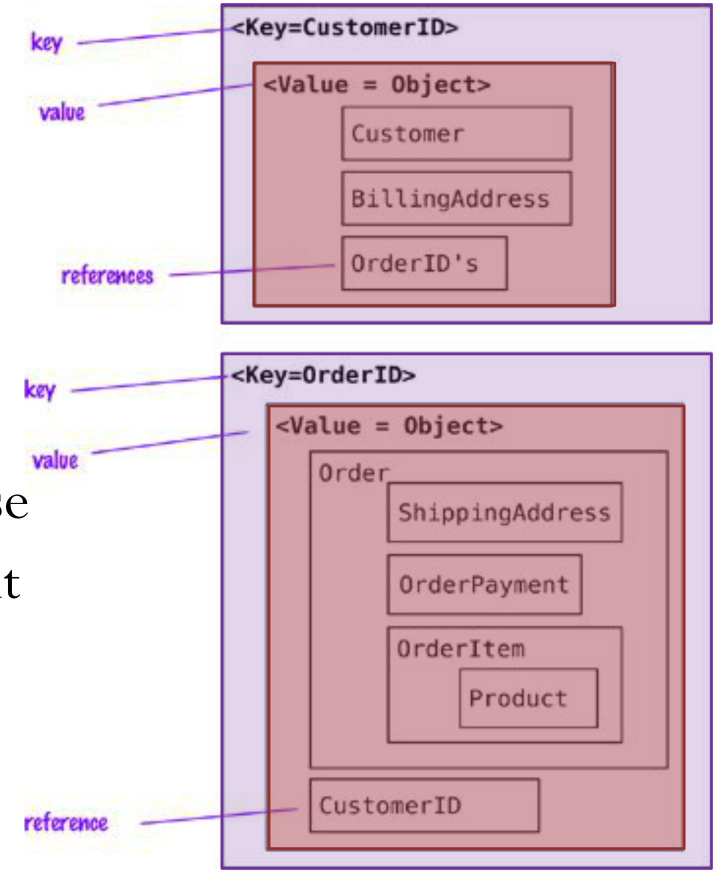
Transactions?

- Relational databases do have ACID transactions!
- Aggregate-oriented databases:
  - don’t have ACID transactions that span multiple aggregates
  - they support atomic manipulation of a single aggregate at a time
- Part of the consideration for deciding how to aggregate data



Key-Value Databases

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model:
  - A set of <key,value> pairs
  - Value: an aggregate instance
- The aggregate is opaque to the database
  - just a big blob of mostly meaningless bit
- Access to an aggregate:
  - lookup based on its key



Popular key-value databases

MEMCACHED

redis

amazon DynamoDB

Azure Cosmos DB

riak

hazelcast

EHCACHE

ORACLE NOSQL DATABASE

Document databases

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model:
  - A set of <key,document> pairs
  - Document: an aggregate instance
- Structure of the aggregate visible
  - limits on what we can place in it
- Access to an aggregate:
  - queries based on the fields in the aggregate

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Popular document databases

mongoDB

Couchbase

CouchDB

Firebase Realtime Database

RAVENDB

MarkLogic™

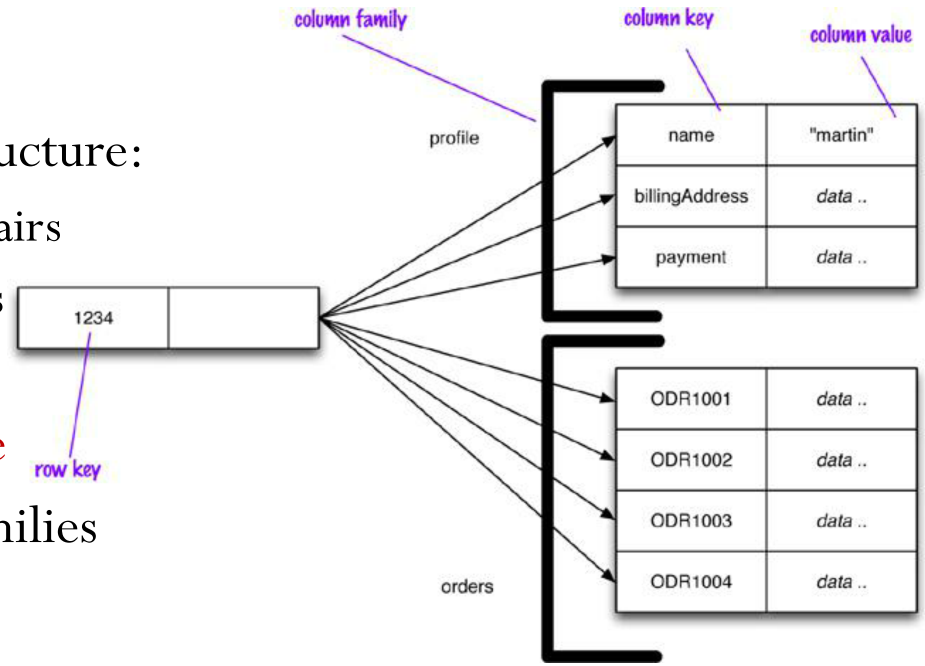
Azure Cosmos DB

Key-Value vs Document stores

- Key-value database
  - A key plus a big blob of mostly meaningless bits
  - We can store whatever we like in the aggregate
  - We can only access an aggregate by lookup based on its key
- Document database
  - A key plus a structured aggregate
  - More flexibility in access
    - we can submit queries to the database based on the fields in the aggregate
    - we can retrieve part of the aggregate rather than the whole thing
  - Indexes based on the contents of the aggregate

Column(-Family) Stores

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model: a two-level map structure:
  - A set of <row-key, aggregate> pairs
  - Each aggregate is a group of pairs <column-key,value>
- Structure of the aggregate visible
- Columns can be organized in families
  - Data usually accessed together
- Access to an aggregate:
  - accessing the row as a whole
  - picking out a particular column



Properties of Column Stores

- Operations also allow picking out a particular column
  - `get('1234', 'name')`
- Each column:
  - has to be part of a single column family
  - acts as unit for access
- You can add any column to any row, and rows can have very different columns
- You can model a list of items by making each item a separate column.
- Two ways to look at data:
  - Row-oriented
    - Each row is an aggregate
    - Column families represent useful chunks of data within that aggregate.
  - Column-oriented:
    - Each column family defines a record type
    - Row as the join of records in all column families

Cassandra

- Skinny row
  - few columns
  - same columns used by many different rows
  - each row is a record and each column is a field
- Wide row
  - many columns (perhaps thousands)
  - rows having very different columns
  - models a list, with each column being one element in that list
- A column store can contain both field-like columns and list-like columns

Popular column stores

APACHE HBASE

cassandra

HYPERTABLE

accumulo™

Google Bigtable

Key Points

- An aggregate is a collection of data that we interact with as a unit.
- Aggregates form the boundaries for ACID operations with the database
- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database
- Aggregates make it easier for the database to manage data storage over clusters
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate
- Aggregate-ignorant databases are better when interactions use data organized in many different formations

Relationships

- Relationship between different aggregates:
  - Put the ID of one aggregate within the data of the other
  - Join: write a program that uses the ID to link data
  - The database is ignorant of the relationship in the data

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
  }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

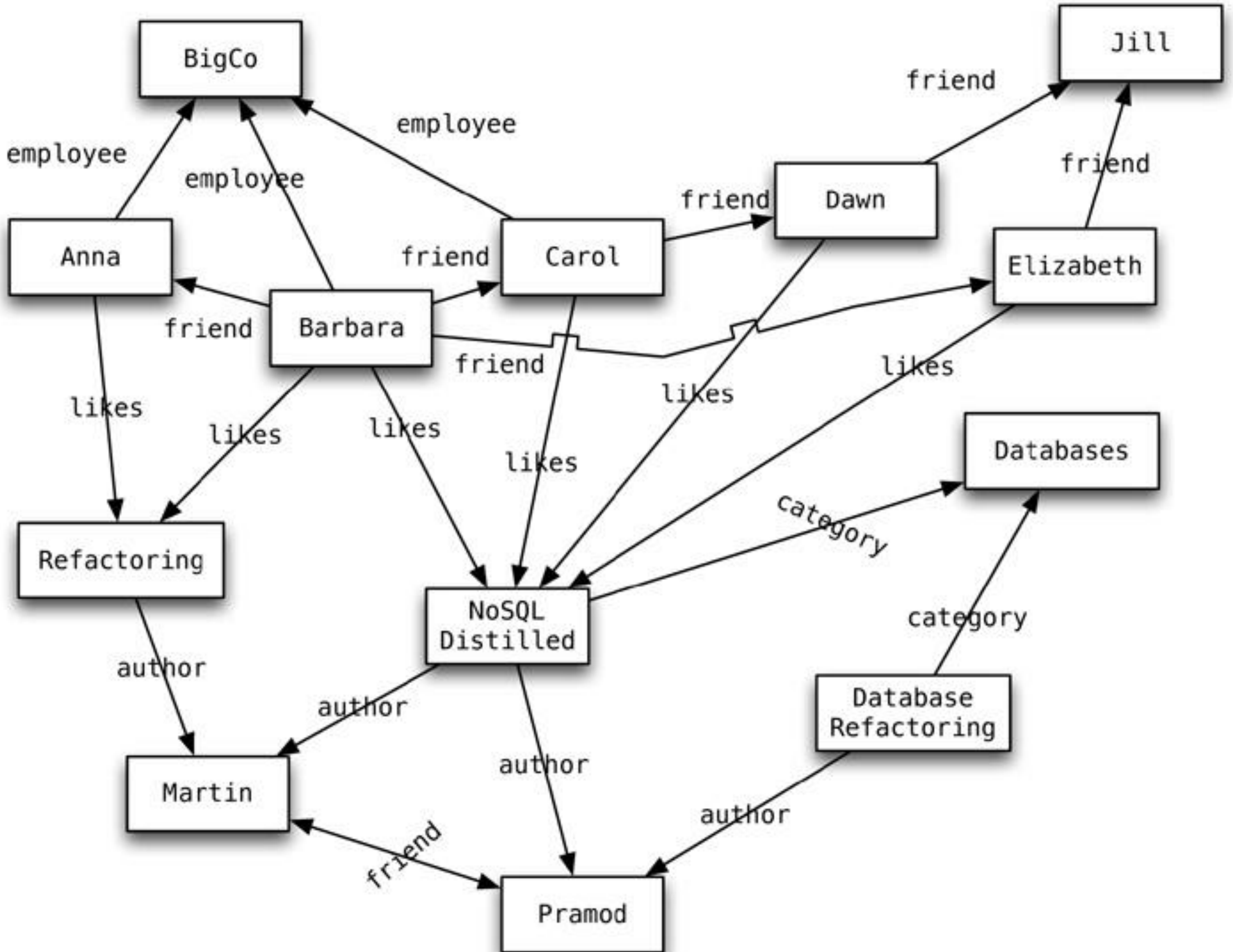
Relationship management

- Many NoSQL databases provide ways to make relationships visible to the database
  - Document stores makes use of indexes
  - Riak (key-value store) allows you to put link information in metadata
- But what about updates?
  - Aggregate-oriented databases treat the aggregate as the unit of data-retrieval.
  - Atomicity is only supported within the contents of a single aggregate.
  - Updates over multiple aggregates at once is a programmer's responsibility!
  - In contrast, relational databases provide ACID guarantees while altering many rows through transactions

Graph Databases

- Graph databases are motivated by a different frustration with relational databases
  - Complex relationships require complex join
- Goal:
  - Capture data consisting of complex relationships
  - Data naturally modelled as graphs
  - Examples: Social networks, Web data, maps, networks.

A graph database



Possible query: “find the authors of books in the Databases category that a friend of mine likes.”

Popular graph databases

neo4j

OrientDB

TITAN

Virtuoso
Universal Server

ArangoDB

AllegroGraph
Franz Inc.

InfiniteGraph
Powered by Objectivity

APACHE
GIRAPH

Data model of graph databases

- Basic characteristic: nodes connected by edges (also called arcs).
- Beyond this: a lot of variation in data models
  - Neo4J stores Java objects to nodes and edges in a schemaless fashion
  - InfiniteGraph stores Java objects, which are subclasses of built-in types, as nodes and edges.
  - FlockDB is simply nodes and edges with no mechanism for additional attributes
- Queries
  - Navigation through the network of edges
  - You do need a starting place
  - Nodes can be indexed by an attribute such as ID.

Graph vs Relational databases

- Relational databases
  - implement relationships using foreign keys
  - joins require to navigate around and can get quite expensive
- Graph databases
  - make traversal along the relationships very cheap
  - performance is better for highly connected data
  - shift most of the work from query time to insert time
  - good when querying performance is more important than insert speed

Graph vs Aggregate-oriented databases

- Very different data models
- Aggregate-oriented databases
  - distributed across clusters
  - simple query languages
  - no ACID guarantees
- Graph databases
  - more likely to run on a single server
  - graph-based query languages
  - transactions maintain consistency over multiple nodes and edges

Schemaless Databases

- Key-value store allows you to store any data you like under a key
- Document databases make no restrictions on the structure of the documents you store
- Column-family databases allow you to store any data under any column you like
- Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish

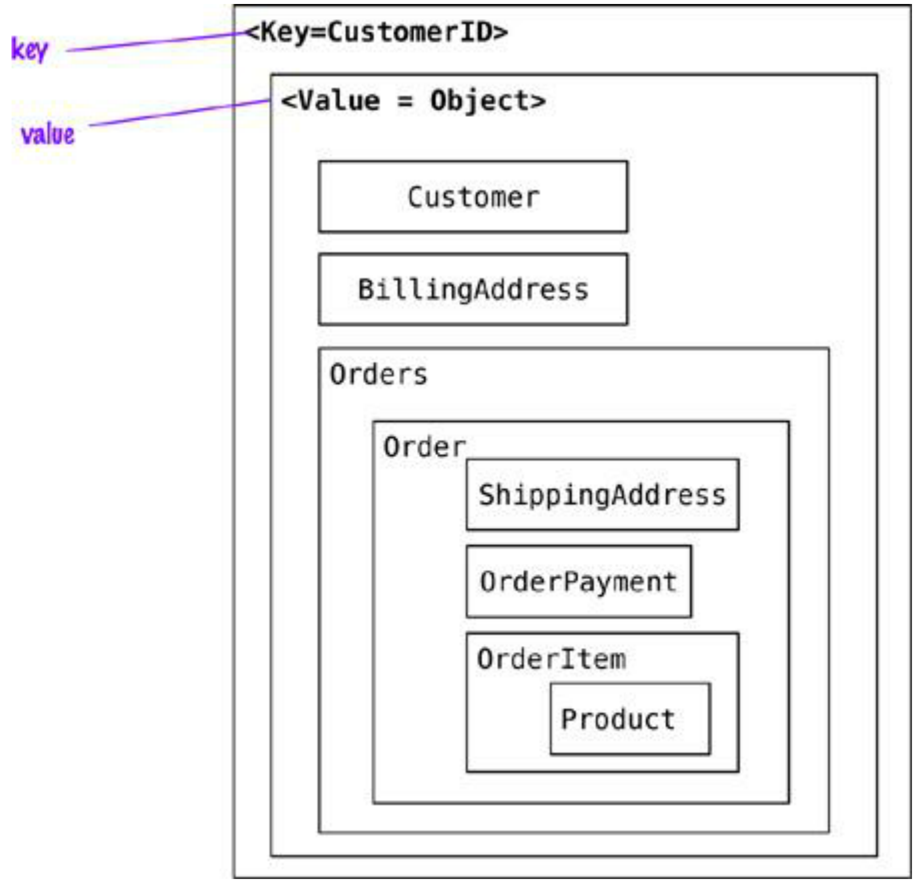
Pros and cons of schemaless data

- Pros:
  - More freedom and flexibility
  - You can easily change your data organization
  - You can deal with non-uniform data
- Cons:
  - A program that accesses data:
    - almost always relies on some form of implicit schema
    - it assumes that certain fields are present
  - The implicit schema is shifted into the application code that accesses data
    - To understand what data is present you have look at the application code
  - The schema cannot be used to:
    - decide how to store and retrieve data efficiently
    - ensure data consistency
  - Problems if multiple applications, developed by different people, access the same database.

Materialized Views

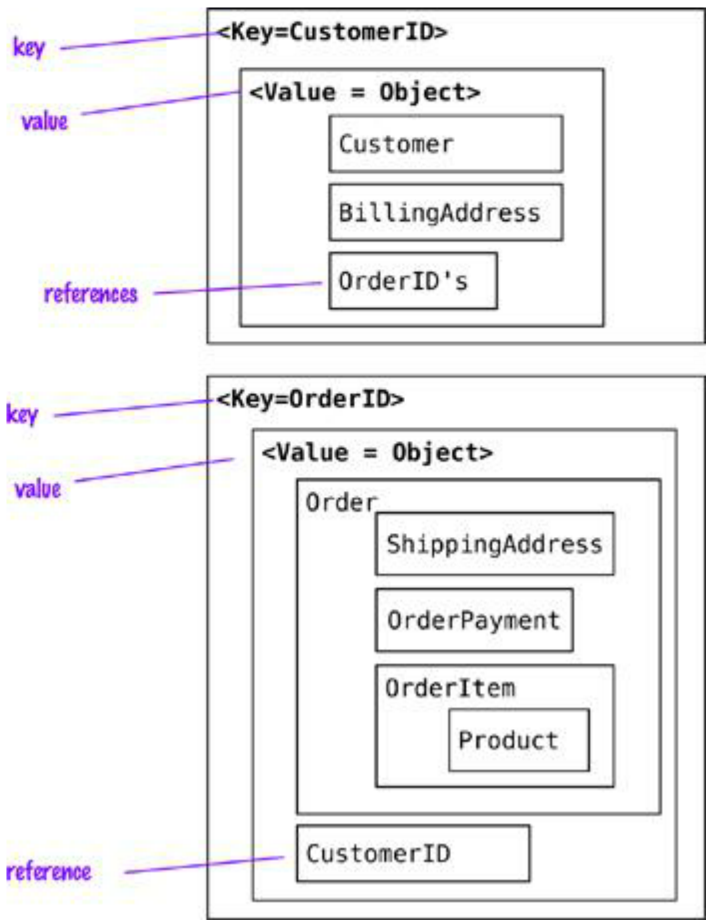
- A relational view is a table defined by computation over the base tables
- Materialized views: computed in advance and cached on disk
- NoSQL databases:
  - do not have views
  - have precomputed and cached queries usually called “materialized view”
- Strategies to building a materialized view
  - Eager approach
    - the materialized view is updated at the same time of the base data
    - good when you have more frequent reads than writes
  - Detached approach
    - batch jobs update the materialized views at regular intervals
    - good when you don’t want to pay an overhead on each update

Data Accesses in key-value store



The application can read all customer's information by using the key

Splitting aggregates




```
# Customer object
{
"customerId": 1,
"customer": {
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [{"type": "debit","ccinfo": "1000-1000-1000-1000"}],
  "orders":[{"orderId":99}]
  }
}
```

```
# Order object
{
"customerId": 1,
"orderId": 99,
"order":{
  "orderDate":"Nov-20-2011",
  "orderItems":[{"productId":27, "price": 32.45}],
  "orderPayment":[{"ccinfo":"1000-1000-1000-1000",
          "txnId":"abelif879rft"}],
  "shippingAddress":{"city":"Chicago"}
  }
}
```

We can now find the orders independently from the Customer, and with the orderID reference in the Customer we can find all Orders for the Customer.

Aggregates for analytics

- A view may store which Orders have a given Product in them
- Useful for Real Time Analytic

```
{
"itemid":27,
"orders":{99,545,897,678}
}
{
"itemid":29,
"orders":{199,545,704,819}
}
```

Data Accesses in document stores

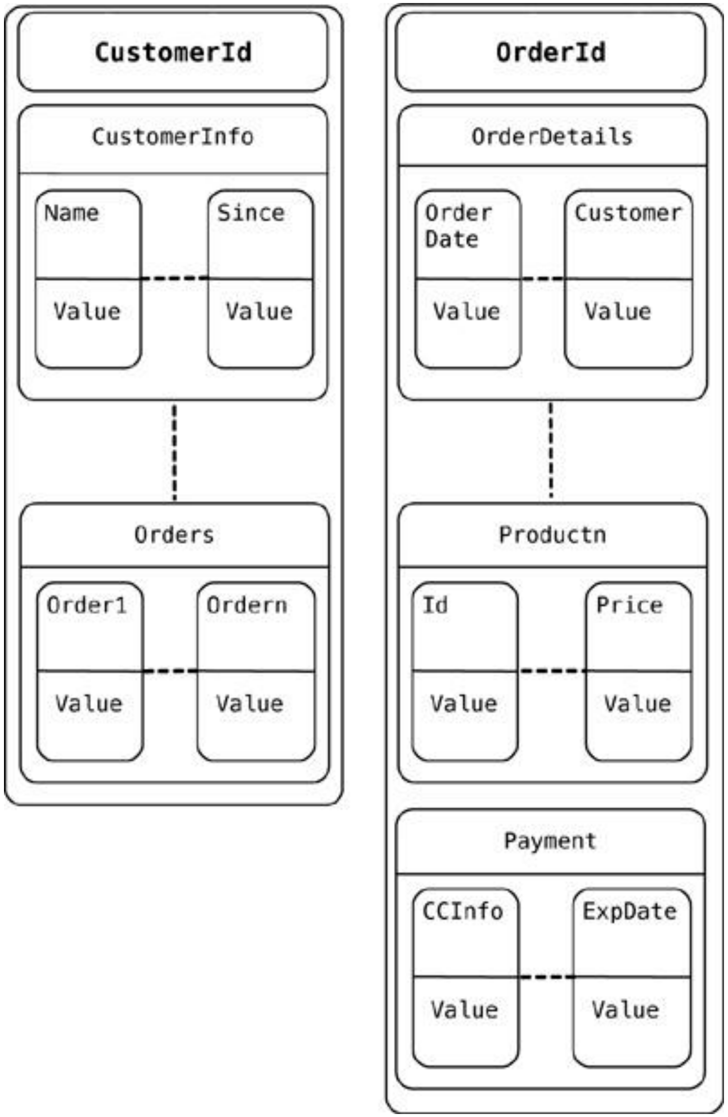
- We can query inside documents:
  - “find all orders that include the Refactoring Databases product”
- Removing references to Orders from the Customer object is possible
- We do not need to update the Customer object when new orders are placed by the Customer

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

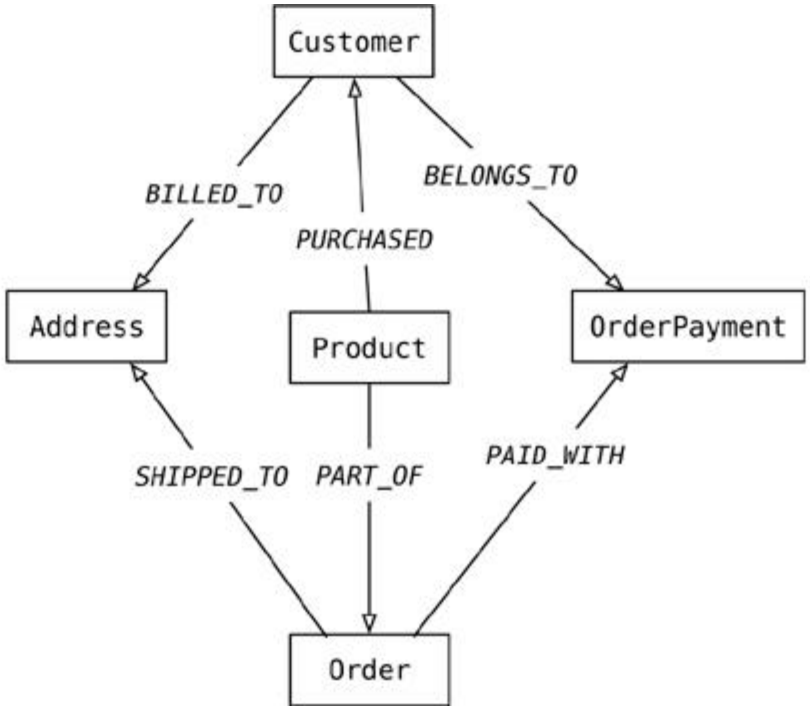
Data Accesses in column-family stores

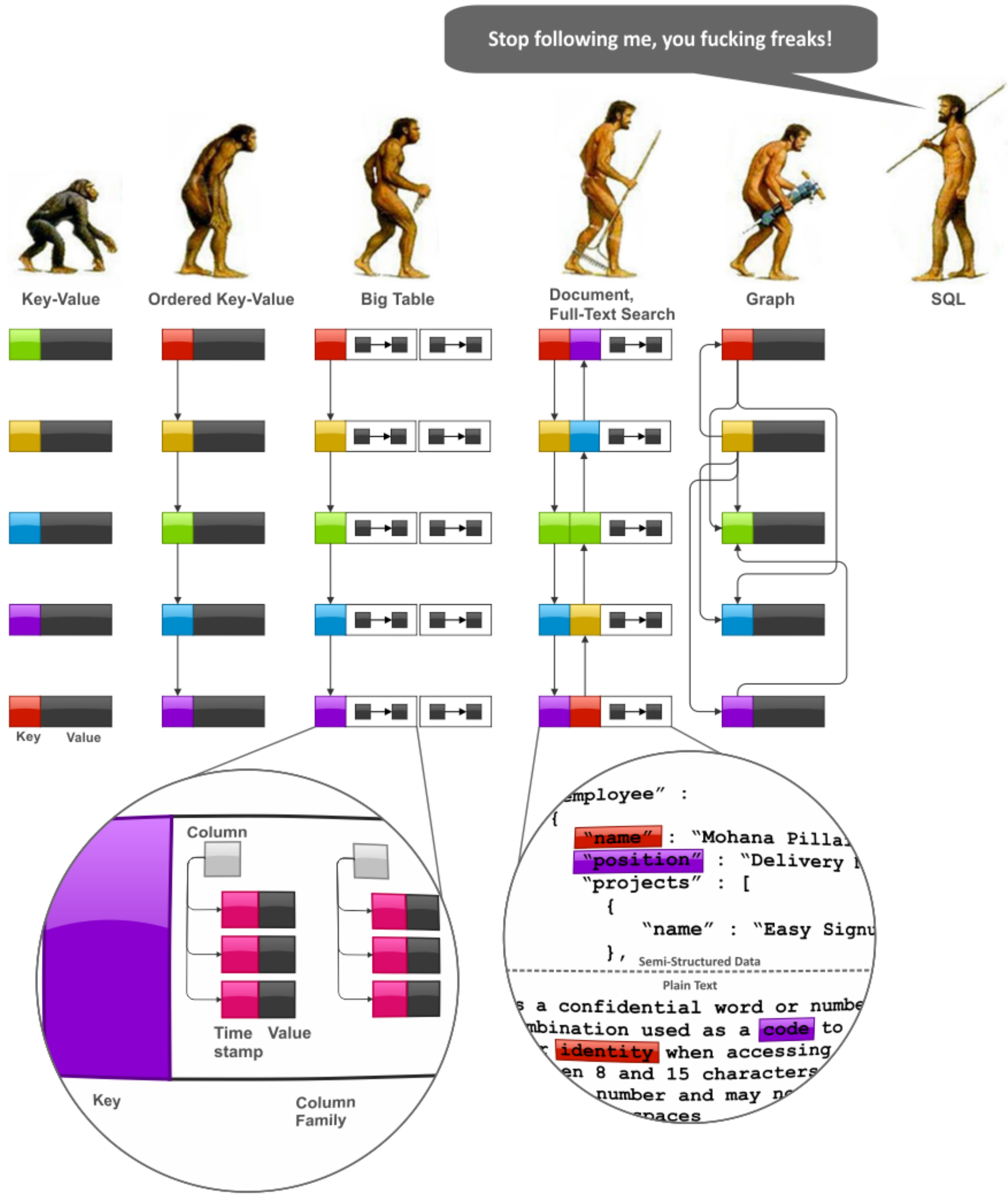
- We can query inside rows:
  - "find all orders whose price is greater than 20$"
- The columns are ordered
- We can choose columns that are frequently used so that they are fetched first
- Splitting data in different column-family families can improve performance



Data Accesses in graph databases

- We start from a (set of) node(s)
- Each node has independent relationships with other nodes
- The relationships have names
- Relationship names let you traverse the graph.





Stop following me, you fucking freaks!
Key-Value
Ordered Key-Value
Big Table
Document, Full-Text Search
Graph
SQL
Key
Value
Column
Time stamp
Value
Key
Column Family
Semi-Structured Data
Plain Text


The continued evolution of Databases
SQL
NoSQL

Key Points

- Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships.
- Graph databases organize data into node and edge graphs; they work best for data that has complex relationship structures.
- Schemaless databases allow you to freely add fields to records, but there is usually an implicit schema expected by users of the data.
- Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates. This is often done with MapReduce-like computations.