Understanding NoSQL

CPS352: Database Systems

Simon Miner
Gordon College
Last Revised: 11/29/12

Agenda

- Check-in
- Why NoSQL?
- NoSQL Data Models
- Related Issues
- Homework 7

Check-in

Why NoSQL?

Pros and Cons of Relational Databases

- Advantages
  - Data persistence
  - Concurrency – ACID, transactions, etc.
  - Integration across multiple applications
  - (Mostly) Standard Model – tables and SQL
- Disadvantages
  - Impedance mismatch
  - Integration databases vs. application databases
  - Not designed for clustering

Impedance Mismatch

- Different representations of data when it is in the RDBMS vs. in memory
  - In-memory data structures use lists, dictionaries, nested and hierarchical data structures
  - Relational database only stores atomic values
    - No lists or nested records
  - Translating between these representations can be costly and confusing
    - Limits the productivity of application developers
- Object-relational mapping (ORM) can help with this
  - Abstraction can lead to neglect of query performance tuning

Impedance Mismatch Example

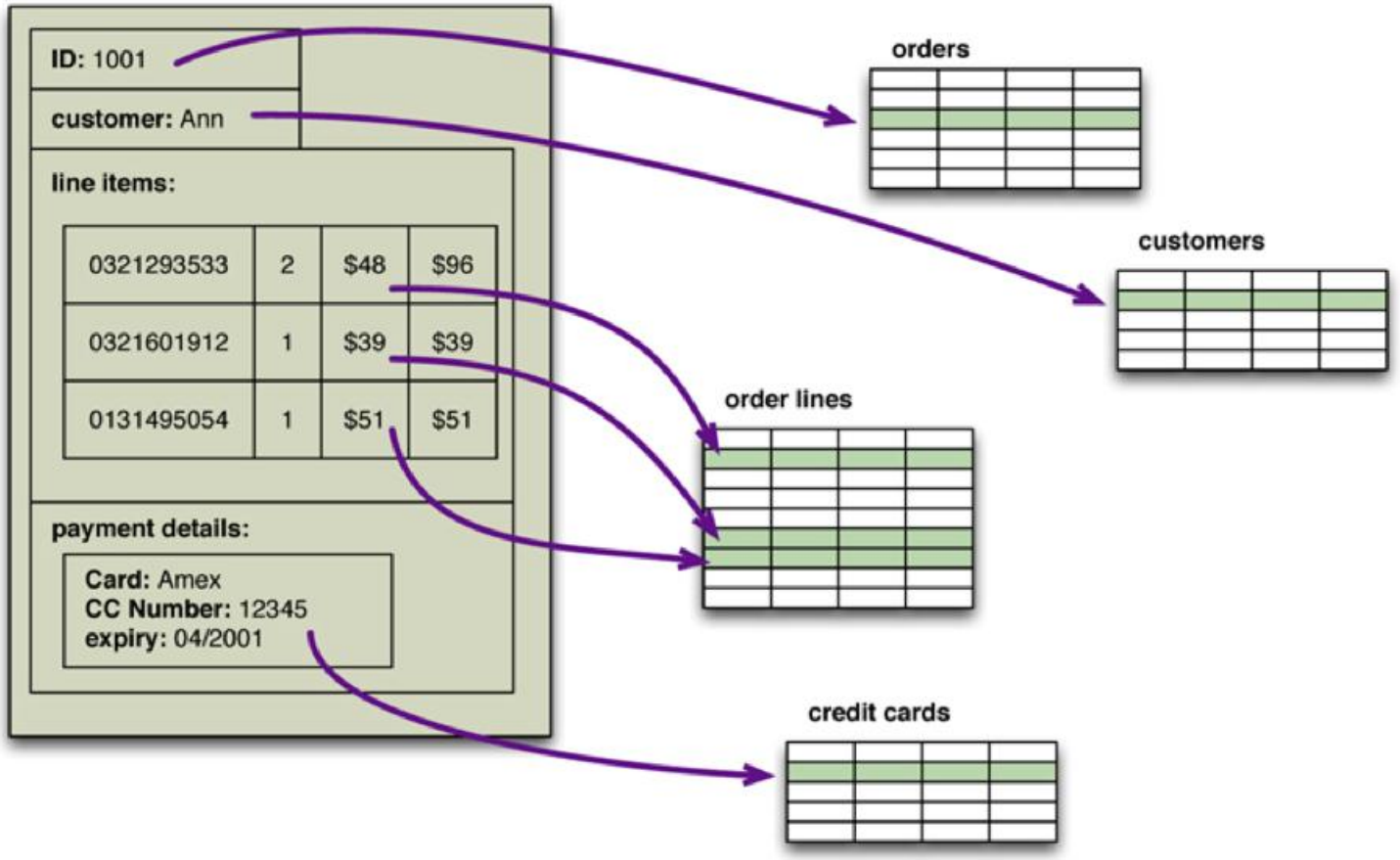

Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

Integration vs. Application Databases

- Integration databases support multiple applications
  - Can be problematic if the applications have very different needs and are maintained by separate teams
- SQL can be limiting as the only shared layer
  - Web services have become a more flexible alternative
- Application databases are simpler to deal with
  - Don't need to worry about the world outside of an application needing to know how its data is structured
  - Security and flexibility decrease in priority

The Need for Clusters

- The Internet created the need to store and process huge amounts of data
  - Relational databases can scale “up” (bigger machine) , but not “out” (many machines) as well
    - Disk subsystem remains a single point of failure
    - Distributing/fragmenting/sharding data is complicated
    - High licensing costs for many database machines and CPUs
- Large web companies began developing their own alternative technologies to deal with these issues
  - Google’s BigTable and Amazon’s Dynamo
  - Issues addressed by these solutions have become relevant to smaller companies wanting to capture and analyze lots of data

The Emergence of NoSQL

- NoSQL first used as a name for an open source relational database released in the late 1990's
- Term as it is used today was a hastily-chosen Twitter hash tag for a conference meet-up on the topic in 2009
- No official general definition for *NoSQL*, but common characteristics include:
  - Does not use the relational model (mostly)
  - Generally open source projects (currently)
  - Driven by the need to run on clusters
  - Built for the need to run 21st century web properties
  - Schema-less
- More of a movement than a technology
  - Relational databases are not going away
  - *Polyglot persistence* – use the type of data store most appropriate for the situation

NoSQL Data Models

Aggregate Data Models

- *Aggregate* – a collection of related objects treated as a unit
  - Particularly for data manipulation and consistency management
- *Aggregate-oriented database* – a database comprised of aggregate data structures
  - Supports atomic manipulation of a single aggregate at a time
  - Good for use in clustered storage systems (scaling out)
    - Aggregates make natural units for replication and fragmentation/sharding
  - Aggregates match up nicely with in-memory data structures
  - Use a key or ID to look up an aggregate record
- An *aggregate-ignorant* data model has no concept of how its components can aggregate together
  - Good when data will be queried in multiple ways
  - Not so good for clusters
    - Need to minimize data accesses, and including aggregates in the data helps with this

Aggregate Database Example: An Initial Relational Model

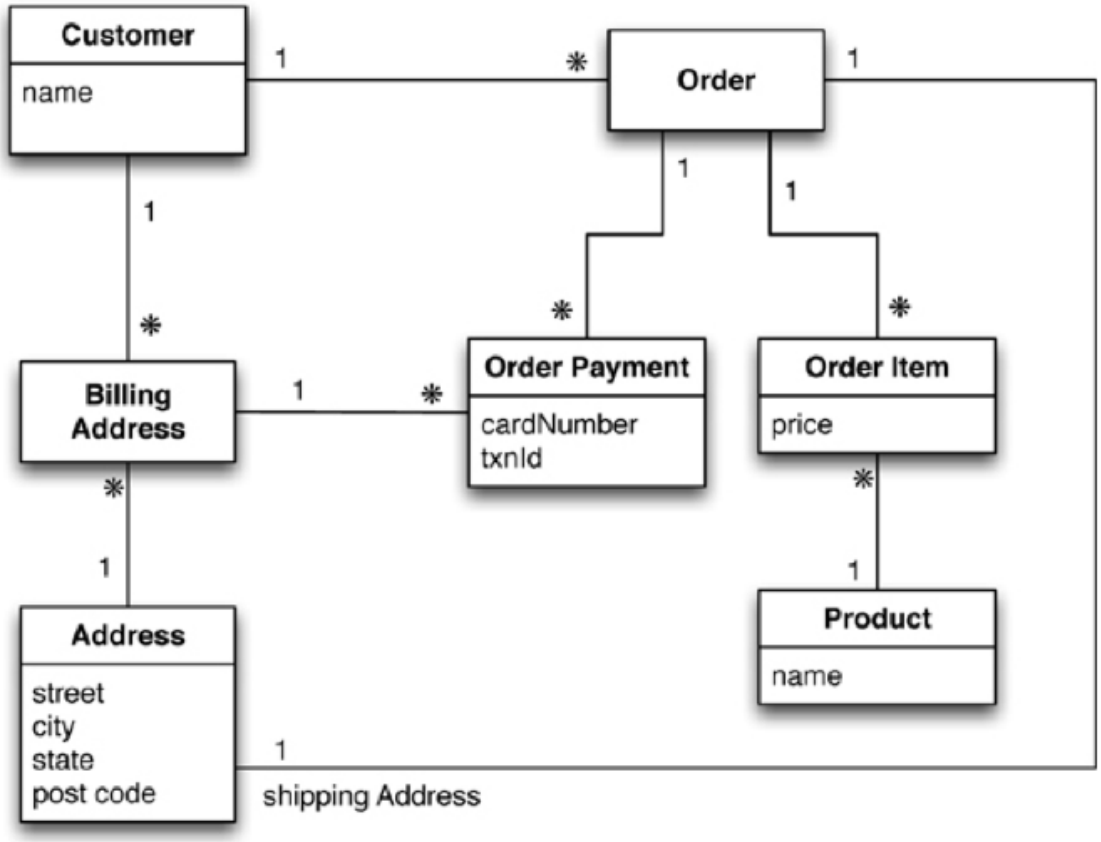

Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

Aggregate Database Example: An Aggregate Data Model

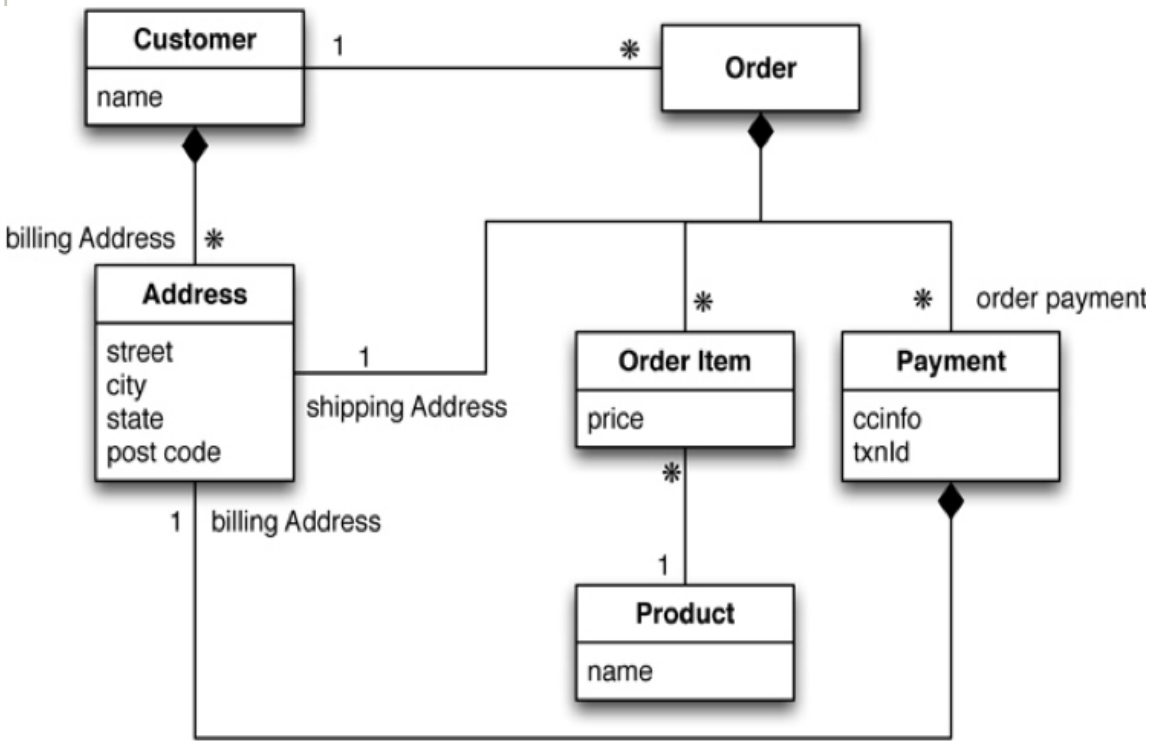

Figure 2.3. An aggregate data model

```
// 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"}
  }
  ],
}
```

Aggregate Database Example: Another Aggregate Model

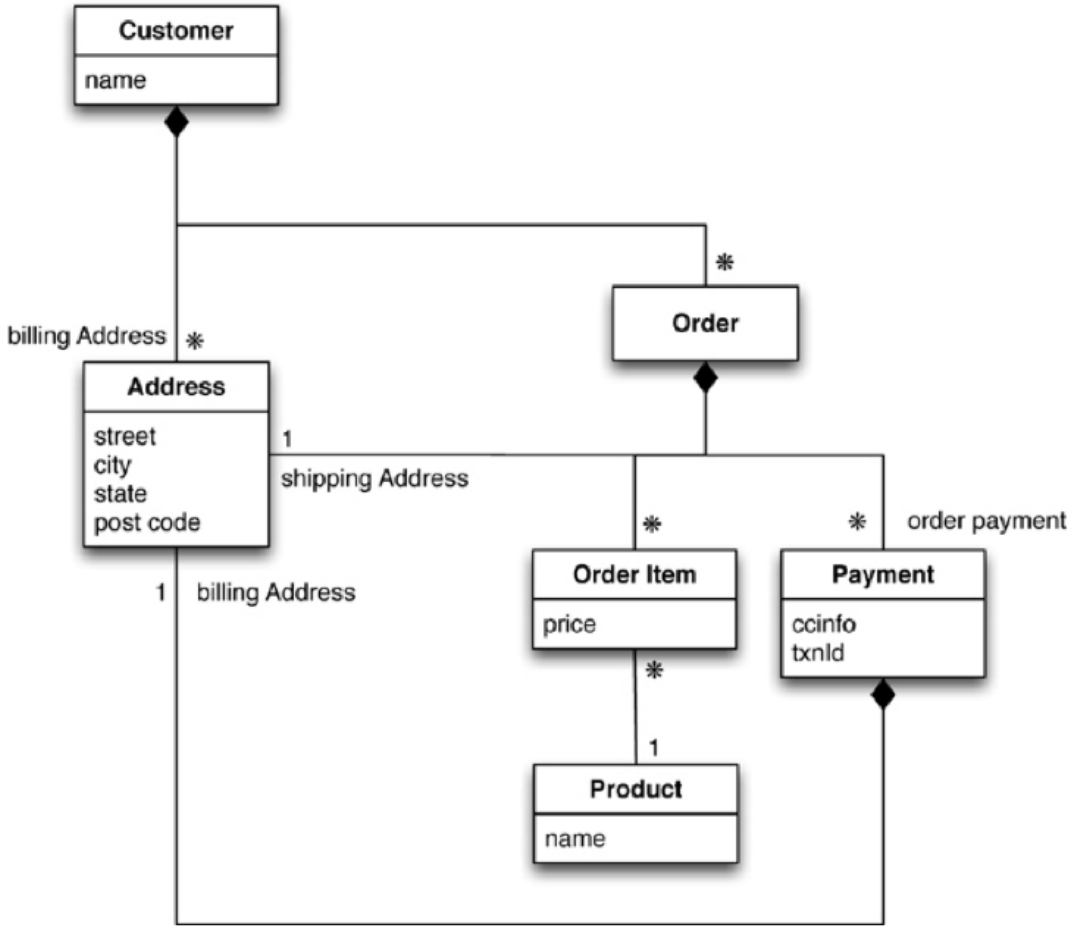

Figure 2.4. Embed all the objects for customer and the customer's orders

```
// 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"}
    }],
  }]
}
}
```

Aggregate-Oriented Databases

- Key-value databases
  - Stores data that is opaque to the database
    - The database does cannot see the structure of records
    - Application needs to deal with this
  - Allows flexibility regarding what is stored (i.e. text or binary data)
- Document databases
  - Stores data whose structure is visible to the database
    - Imposes limitations on what can be stored
    - Allows more flexible access to data (i.e. partial records) via querying
- Both key-value and document databases consist of aggregate records accessed by ID values
- Column-family databases
  - Two levels of access to aggregates (and hence, two pars to the "key" to access an aggregate's data)
    - ID – to look up aggregate record
    - Column name – either a label for a value (name) or a key to a list entry (order id)
  - Columns are grouped into column families

Column-Family Database Example

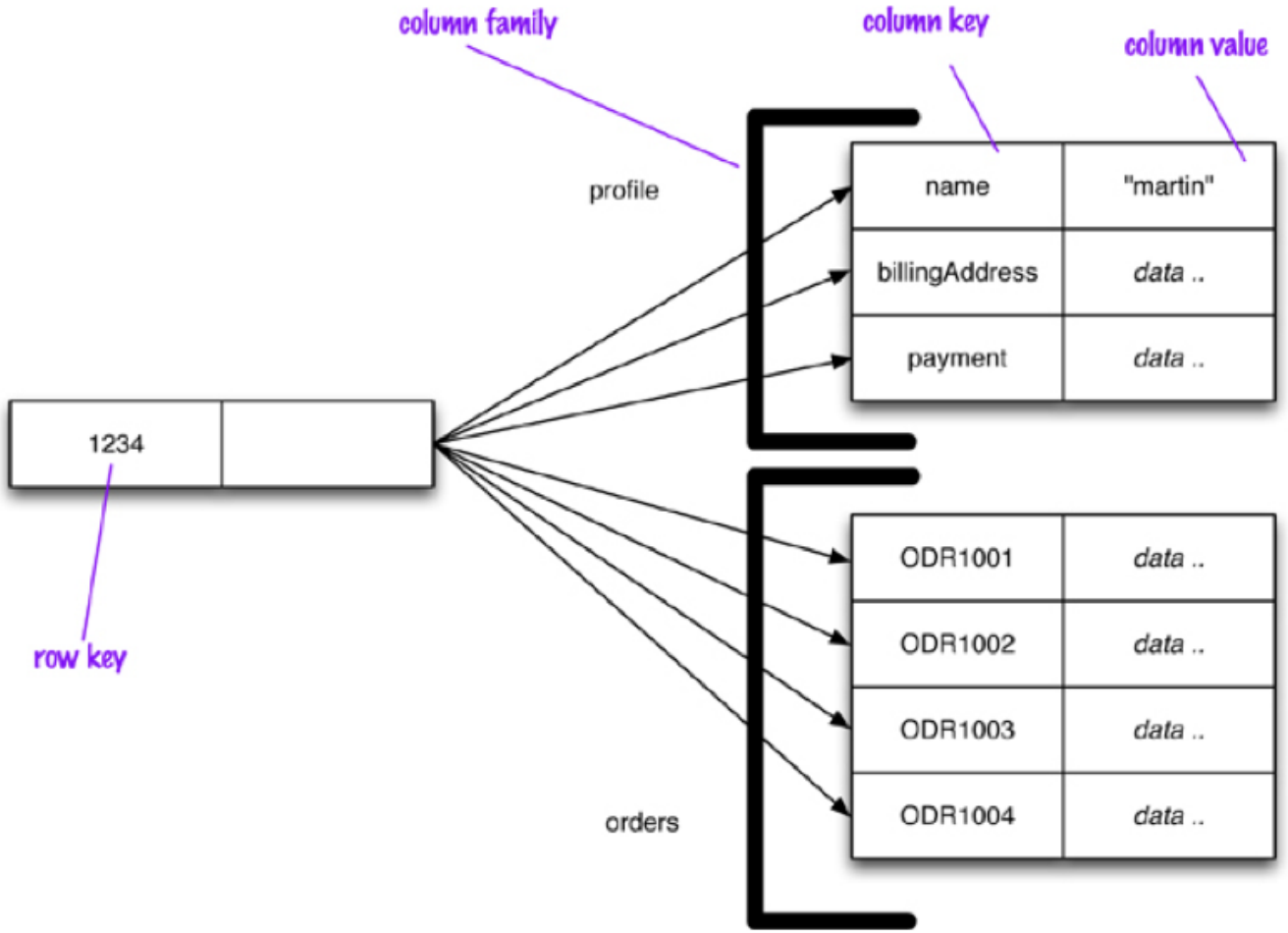

Figure 2.5. Representing customer information in a column-family structure

Relationships

- Aggregates contain ID attributes to related aggregates
  - Require multiple database accesses to traverse relationships
    - One to lookup ID(s) of related aggregate(s) in main aggregate
    - One to retrieve each of the related aggregates
  - Many NoSQL databases provide mechanisms to make relationships visible to the database (to make link-walking easier)
- Updates to relationships require the application to maintain consistency since atomicity is limited to each aggregate
- Aggregate databases become awkward when it is necessary to navigate around many aggregates
- Graph databases – small nodes connected by many edges
  - Make navigating complex relationships fast
    - Linking nodes is done at time of insert, and not at query time

Graph Database Example

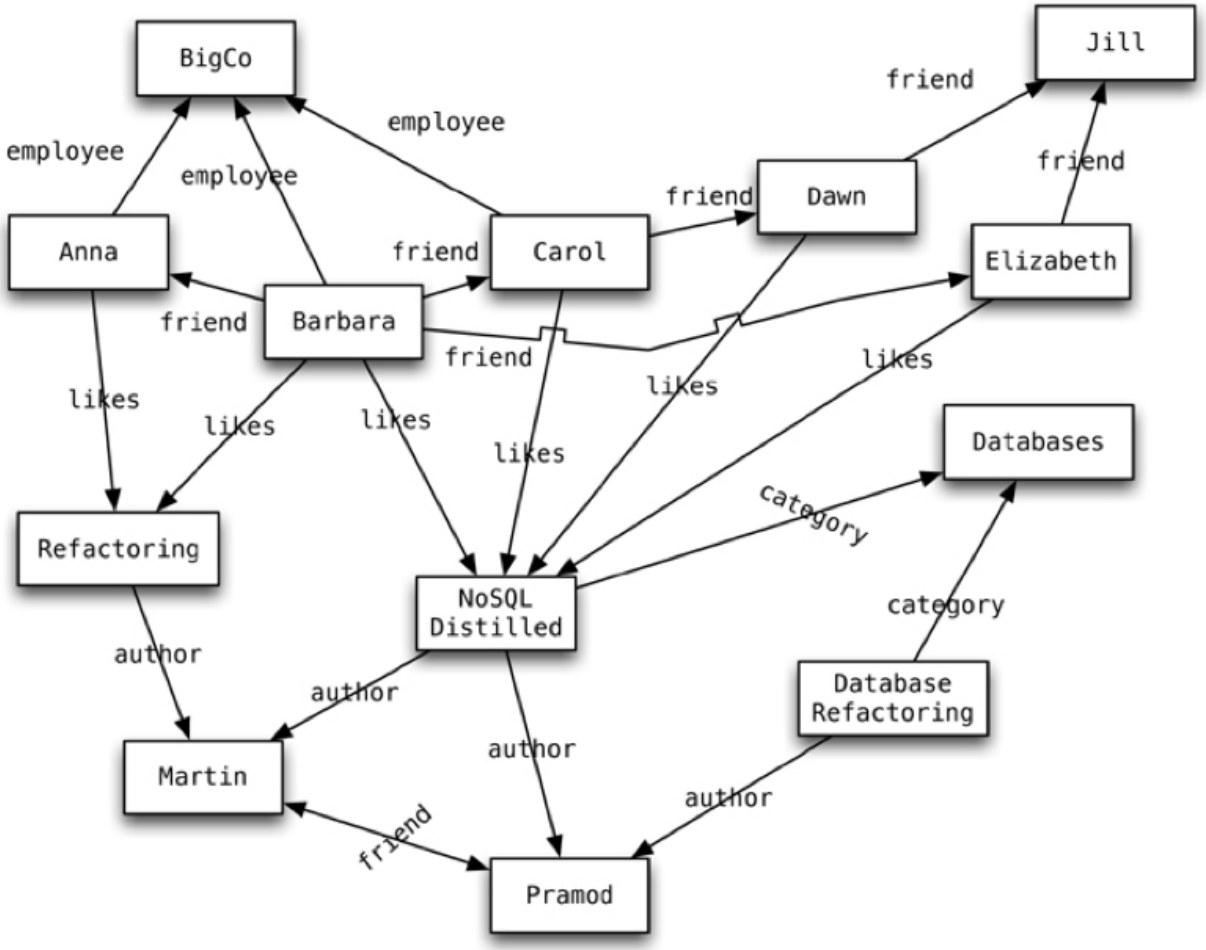

Figure 3.1. An example graph structure

Schema-less Databases

- Common to all NoSQL databases – also called *emergent schemas*
- Advantages
  - No need to predefine data structure
  - Easy to change structure of data as time passes
  - Good support for *non-uniform data*
- Disadvantages
  - Potentially inconsistent names and data types for a single value
    - Example: quantity, Quantity, QUANTITY, qty, count, quanity …
    - Example: 5, 5.0, five, V …
    - The database does not enforce these things because it has no knowledge of the *implicit schema*
  - Management of the implicit schema migrates into the application layer
    - Need to look at code to understand what data and structure is present
      - No standard location or method for implementing the logic to do this
    - What do you do if multiple applications need access to the database?

Materialized Views

- Querying across aggregates is expensive
  - Example: database with customer aggregates containing orders – efficient customer-level queries
    - Inefficient to query across orders (i.e. tally data from orders placed in the last week)
- NoSQL databases can pre-compute expensive query results and store them in *materialized views*
  - Term borrowed from relational databases – a view that is cached
  - Enables faster access of data organized differently from primary aggregates
- Keeping materialized views up-to-date
  - Eager approach – update view with the base data
    - Good for frequent reads of view that needs to be kept fresh
  - Regular batch of view updates

Related Issues

Distributed Databases and Consistency with NoSQL

Version Stamps

Map-Reduce Pattern

Distribution Models

- Single server – simplest model, everything on one machine (or *node*)
- *Sharding* (fragmentation) – storing data (aggregates) across multiple nodes
  - *Auto-sharding* -- some NoSQL databases handle the logistics of sharding so that the application does not have to
- Replication – duplicate data (aggregates) over multiple nodes
  - Master-slave (primary copy) replication -- one master responsible for updates, one or more slaves to support reads
  - Peer-to-peer (multi-master) replication
    - Each node does reads and writes, and communicates its changes to other nodes
      - Eliminates any one master as a single point of failure
    - Drawbacks include complex synchronization system and inconsistency issues
      - Write-write conflicts – when two users update the same data item on separate nodes

Consistency

- Update consistency – ensuring serial database changes
  - *Pessimistic* approach – prevents conflicts from occurring (i.e. locking)
  - *Optimistic* approach – detects conflicts and sorts them out (i.e. validation)
    - Conditional update – just before update, check to see if the value has changed since last read
    - Write-write conflict resolution – automatically or manually merge the updates
  - Trade-off between safety and "liveness" (responsiveness)
- Read consistency – ensuring users read the same value for data at a given time
  - *Logical consistency* vs. *replication consistency*
  - *Sticky sessions* (session affinity) – assign a session to a given database node for all of its work to ensure *read-your-writes consistency*

Diluting the ACID

- Relaxed consistency
  - CAP Theorem – pick two of these three
    - Consistency
    - Availability – ability to read and write data to a node in the cluster
    - Partition tolerance – cluster can survive network breakage that separates it into multiple isolated partitions
  - If there is a network partition, need to trade off availability of data vs. consistency
    - Depending on the domain, it can be beneficial to balance consistency with latency (performance)
    - BASE – Basically Available, Soft state, Eventual consistency
- Relaxed durability
  - Replication durability – what happens if a replica is not available to receive updates, but still servicing traffic?
  - Do not necessarily need to contact all replicas to preserve strong consistency with replication; just a large enough quorum.

Version Stamps

- Provide a means of detecting concurrency conflicts
  - Each data item has a version stamp which gets incremented each time the item is updated
  - Before updating a data item, a process can check its version stamp to see if it has been updated since it was last read
- Implementation methods
  - Counter – requires a single master to "own" the counter
  - GUID (Guaranteed Unique ID) – can be computed by any node, but are large and cannot be compared directly
  - Hash the contents of a resource
  - Timestamp of last update – node clocks must be synchronized
- Vector stamp – set of version stamps for all nodes in a distributed system
  - Allows detection of conflicting updates on different nodes

Map-Reduce

- Design pattern to take advantage of clustered machines to do processing in parallel
  - While keeping as much work and data as possible local to a single machine
- Map function
  - Takes a single aggregate record as input
  - Outputs a set of relevant key-value pairs
    - Values can be data structures
  - Each instance of the map function is independent from all others
    - Safely parallelizable
- Reduce function
  - Takes multiple map outputs with the same key as input
  - Summarizes (or *reduces*) there values to a single output
- Map-reduce framework
  - Arranges for map function to be applied to pertinent documents on all nodes
  - Moves data to the location of the reduce function
  - Collects all values for a single pair and calls the reduce function on the key and value collection
  - Programmers only need to supply the map and reduce functions

Map-Reduce Example (Map)

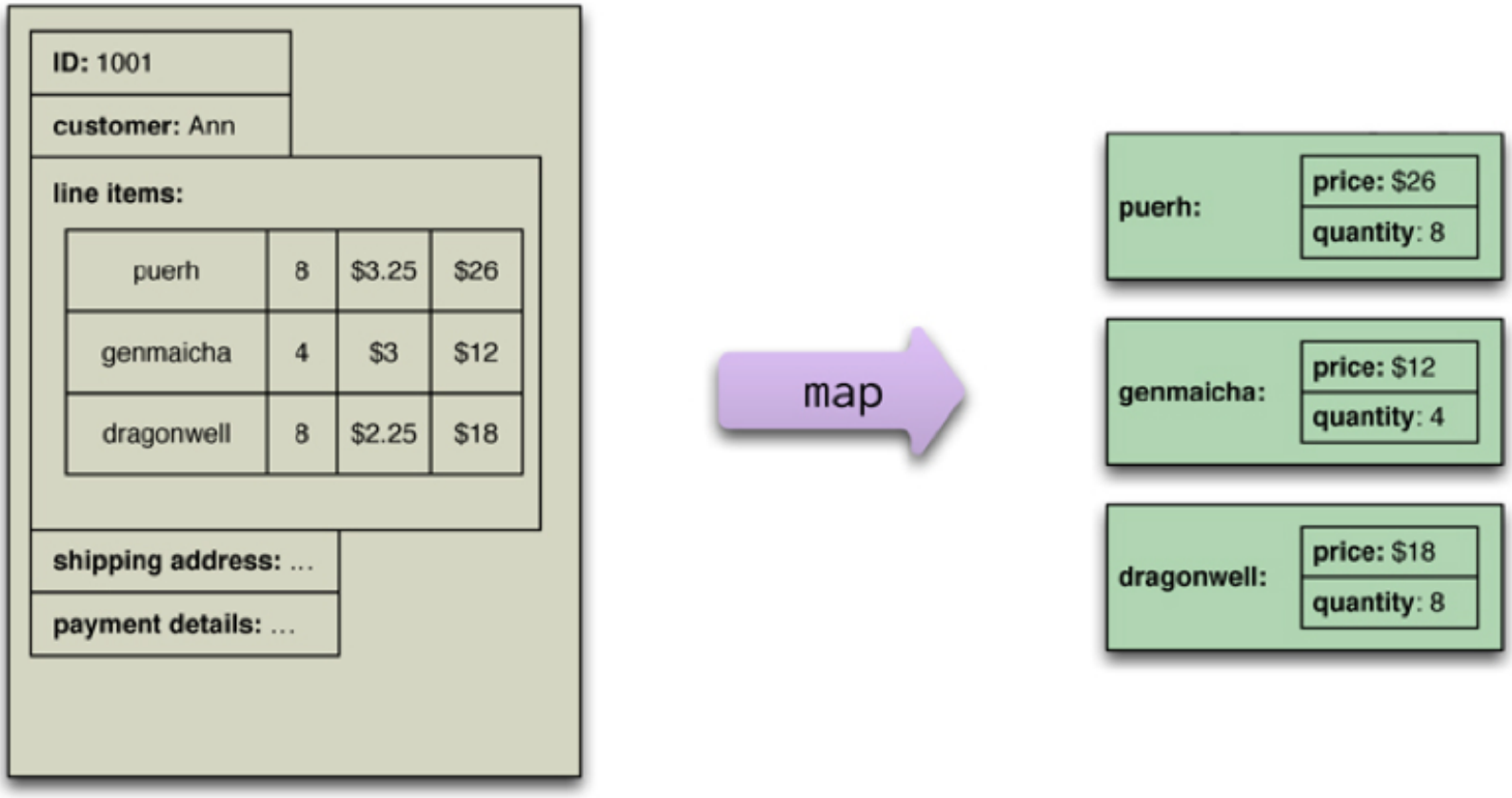

Figure 7.1. A map function reads records from the database and emits key-value pairs.

Map-Reduce Example (Reduce)

Figure 7.2. A reduce function takes several key-value pairs with the same key and aggregates them into one.

Partitioning, Combining, and Composing

- Reduce operations use values from a single key
  - Partitioning by key allows for parallel reduce work
- *Combinable reducer* -- Reducers that have the same form for input and output can be combined into pipelines
  - Further improves parallelism and reduces the amount of data to be transferred
- Map-reduce compositions
  - Can be composed into pipelines in which the output of one reduce is the input to another map
  - Can be useful to store result of widely-used map-reduce calculation
    - Saved results can sometimes be updated incrementally
      - For additive combinable reducers, the existing result can be combined with new data

Reduce Partitioning Example

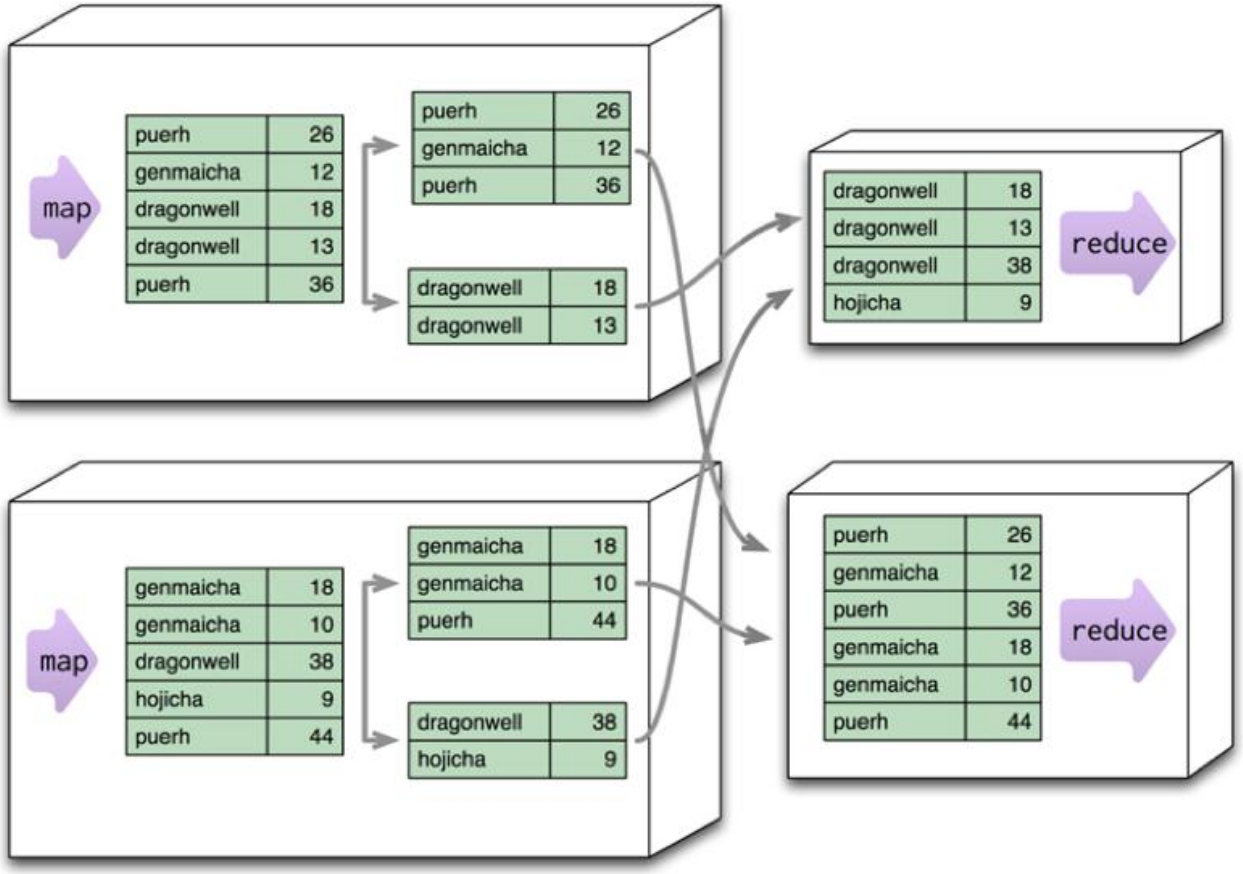

Figure 7.3. Partitioning allows reduce functions to run in parallel on different keys.

Combinable Reducer Example

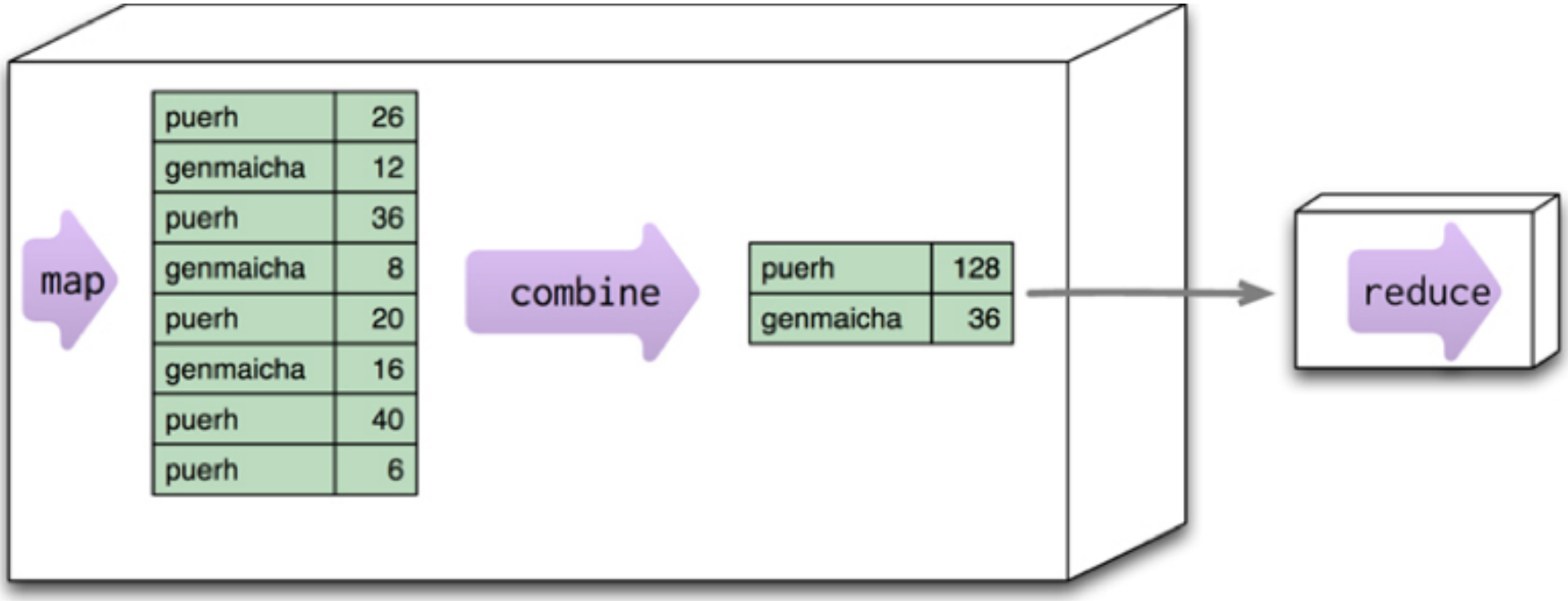

Figure 7.4. Combining reduces data before sending it across the network.

Multi-Stage Map-Reduce Example

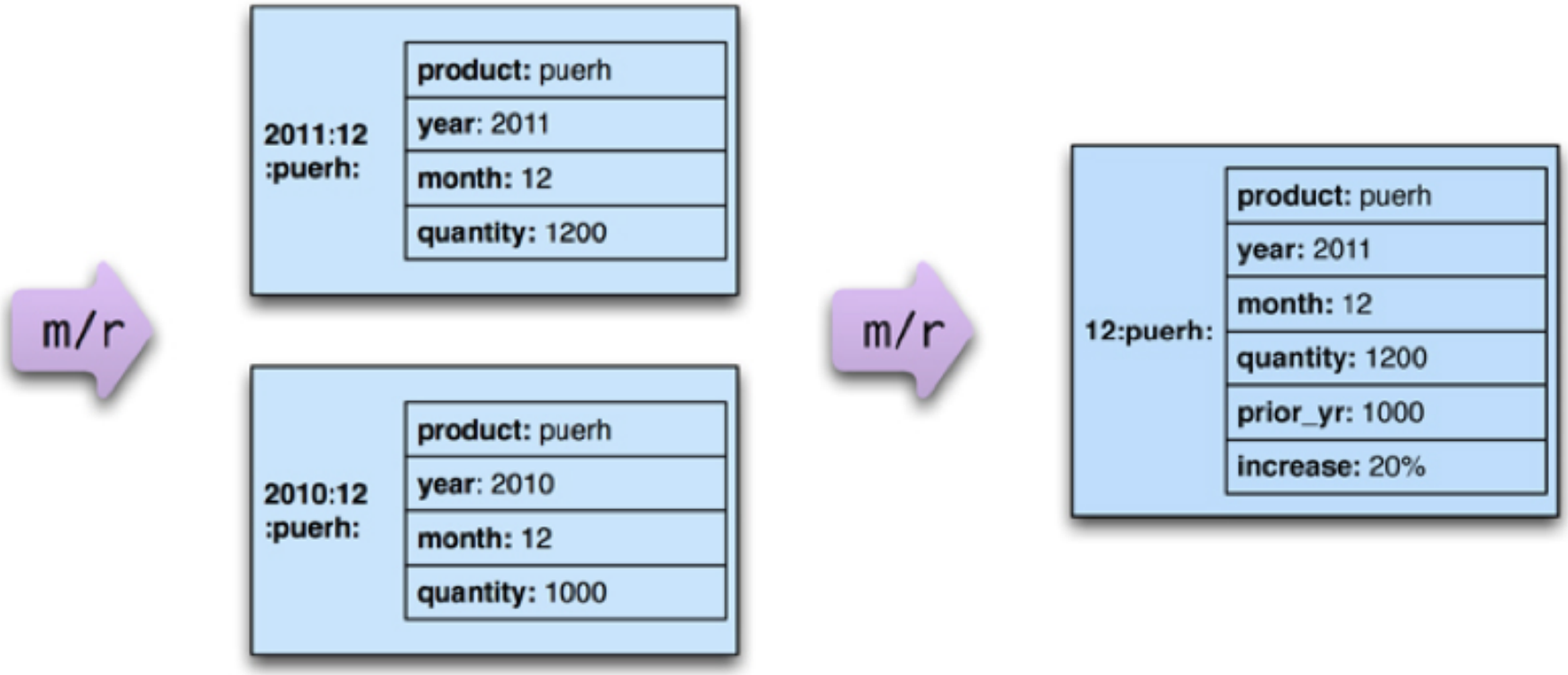

Figure 7.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

Homework 7