Using NoSQL

CPS352: Database Systems

Simon Miner
Gordon College
Last Revised: 12/6/12
Agenda

• Check-in

• NoSQL Databases
  • Aggregate databases – Key-value, document, and column family
  • Graph databases

• Further Matters
  • Schema Migrations
  • Polyglot Persistence
  • When (not) to use NoSQL

• Google Spanner Video
Check-in
NoSQL Databases

Aggregate Databases: Key-value, Document, Column Family
Graph Databases
Key-Value Databases

- Key-value store is a simple hash table
  - Records access via key (ID)
    - Akin to a primary key for relational database records
    - Only / quickest way to access a record
  - *Values* can be of any type -- database does not care
    - Like blob data type in relational database
  - *Bucket* – namespace used to segment keys
    - Shows up as (sometimes implicit) prefix or suffix to key

- Operations
  - Get a value for a given key
  - Set (or overwrite or append) a value for a given key
  - Delete a key and its associated value
Key-Value Database Features

- Consistency only applies in the context of a single key/value pair
  - Need strategy to handle distributed key-value pairs – i.e. newest write wins, all writes reported and client resolves the conflict

- No ACID transactions because of performance requirements over distributed cluster
  - Weaker transaction consistency can be asserted by requiring that a certain number of nodes (quorum) get the write

- Scale by both fragmentation and replication
  - Shard by key values (using a uniform function)
  - Replicas should be available in case a shard fails
    - Otherwise all reads and writes to the unavailable shard fail
Interacting with Key-Value Databases

- Applications can only query by key, not by values in the data

- Design of key is important
  - Must be unique across the entire database
  - Bucket can provide an implicit top-level namespace

- How and what data gets stored is managed entirely at the application level
  - Single key for related data structures
    - Key incorporates identification data (i.e. user_<sessionID>)
      - Data can include various nested data structures (i.e. user data including session, profile, cart info)
    - All data is set and retrieved at once
  - Different kinds of aggregates all stored in one bucket
    - Increases chance of key conflicts (i.e. profile and session data with same ID)
  - Multiple keys for related data structures
    - Key incorporates name of object being stored (i.e. user_<sessionID>_profile)
    - Multiple targeted fetches needed to retrieve related data
    - Decreases chance of key conflicts (aggregates have their own specific namespaces)

- Expiration times can be assigned to key-value pairs (good for storing transient data)
Key-Value Aggregate Examples

Figure 8.1. Storing all the data in a single bucket.

Figure 8.2. Change the key design to segment the data in a single bucket.
Using Key-Value Databases

- Use key-value databases for...
  - Data accessed via a unique key (i.e. session, user profile, shopping cart, etc.)
  - Transient data
  - Caching

- Don’t use key-value databases for...
  - Relationships among data
  - Multi-operation transactions
  - Querying by data (value instead of key)
  - Operations on sets of records
Document Databases

- Store of documents with keys to access them
  - Similar to key-value databases except…
  - Can see and dynamically manipulate the structure of the documents
    - Often structured as JSON (textual) data
    - Each document can have its own structure (non-uniform)
    - Each document is (automatically) assigned an ID value (_id)

- Consistency and transactions apply to single documents

- Replication and sharding are by document

- Queries to documents can be formatted as JSON
  - Able to return partial documents
### Document Database Example

```sql
// in order collection
{
  "customerId": 12345,
  "orderId": 67890,
  "orderDate": "2012-12-06",
  "items": [
    {
      "product": {
        "id": 112233,
        "name": "Refactoring",
        "price": "15.99"
      },
      "discount": "10%"
    },
    {
      "product": {
        "id": 223344,
        "name": "NoSQL Distilled",
        "price": "24.99"
      },
      "discount": "3.00",
      "promo-code": "cybermonday"
    }
  ],
  "discount": "10%"
}
```

<table>
<thead>
<tr>
<th>SQL</th>
<th>Document Database Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>select * from order</td>
<td>db.order.find()</td>
</tr>
</tbody>
</table>
| select * from order where customerId = 12345 | db.order.find({
  "customerId": 12345
}) |
| select orderId, orderDate from order where customerId = 12345 | db.order.find({
  "customerId": 12345,
  "orderId": 1,"orderDate": 1
}) |
| select * from order o join orderItem oi on o.orderId = oi.orderID join product p on oi.productId = p.Id where p.name like '%Refactoring%' | db.order.find({
  "items.product.name": "/Refactoring/"
}) |
Using Document Databases

- Use document databases for…
  - Event logging – central store for different kinds of events with various attributes
  - Content management or blogging platforms
  - Web analytics stores
  - E-commerce applications

- Do not use document databases for…
  - Transactions across multiple documents (records)
  - Ad hoc cross-document queries
Column Family Databases

- Structure of data records
  - Each record indexed by a key
  - Columns grouped into column families (like RDBMS tables)
  - Related columns can also be grouped into super columns
  - Related super columns can be grouped into super column families

- Additional mechanisms to assist with data management
  - Key space – top-level container for a certain kind of data (kind of like a schema in RDBMS)
    - Configuration parameters and operations can apply to a key space
      - i.e. number of replicas, data repair operations
    - Columns are specified when a key space is created, but new ones can be added at any time, to only those rows they pertain to

- Data access
  - Get, set, delete operations
  - Query language (i.e. CQL – Cassandra Query Language)
Column Family Database Example

CREATE COLUMNFAMILY Customer (KEY varchar PRIMARY KEY, name varchar, city varchar, web varchar);

INSERT INTO Customer (KEY,name,city,web) VALUES ('mfowler','Martin Fowler','Boston','www.martinfowler.com');

SELECT * FROM Customer;

SELECT name,web FROM Customer WHERE city='Boston'
Using Column Family Databases

- Use column family databases for…
  - Event logging
  - Content management and blogging platforms
  - Counters
  - Expiring data

- Do not use column family databases for…
  - Systems requiring ACID transactions
  - Systems requiring ad-hoc aggregate queries
**Data Management Scale with Aggregate Databases**

- Different aggregate data models have differing data management capabilities
  - Key-value databases
    - Opaque data store
    - Almost no database involvement with managing data
  - Document databases
    - Transparent data store
    - Some facilities in databases to administer data (partial record queries, indexes)
  - Column family databases
    - Transparent data store and dynamic schema
    - Data management constructs (key spaces, query languages)
  - Relational databases
    - Static uniform schema
    - Database manages the data (integrity constraints, security, etc.)
Graph Databases

- Excel at modeling relationships between entities
- **Terminology**
  - *Node* – an entity or record in the database
  - *Edge* – a directed relationship connecting two entities
    - Two nodes can have multiple relationships between them
  - *Property* – attribute on a node or edge
- **Graphs are queried via *traversals***
  - Traversing multiple nodes and edges is very fast
    - Because relationships are determined when data is inserted into the database
  - Relationships (edges) are persisted just like nodes
    - Not computed at query time (as in relational databases)
Graph Database Example

Figure 11.2. Relationships with properties
Graph Database Features

- Transaction support – graph can only be modified within a transaction
  - No “dangling relationships” allowed
  - Nodes can only be deleted if they have no edges connected to them

- Availability via replication

- Scaling via sharding is difficult since the graph relies heavily on the relationships between its nodes
  - Fragmentation can be done using domain knowledge (i.e. separating relationships by different geographic regions, categories, time periods, etc. – factors don’t get traversed much)
  - Traversal across shards is very expensive
Interacting with Graph Databases

• Web services / REST APIs exposed by the database

• Language-specific libraries provided by the database vendor or community

  // Find the names of people who like NoSQL Distilled
  Node nosqlDistilled = nodeIndex.get("name", "NoSQL Distilled").getSingle();
  relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
  for (Relationship relationship : relationships) {
    likesNoSQLDistilled.add(relationship.getStartNode());
  }

• Query languages – allow for expression of complex queries on the graph
  • Gremlin with Blueprints (JDBC-like) database connectors
  • Cypher (for neo4j)
Graph Database Query Language Example

- A “select” statement in Cypher
  
  START beginingNode = (beginning node specification)
  MATCH (relationship, pattern matches)
  WHERE (filtering condition: on data in nodes and relationships)
  RETURN (What to return: nodes, relationships, properties)
  ORDER BY (properties to order by)
  SKIP (nodes to skip from top)
  LIMIT (limit results)

- Find the names and locations of Barbara’s friends
  
  • Cypher
    
    START barbara = node:nodeIndex(name = "Barbara")
    MATCH (barbara)-[:FRIEND]->(friend_node)
    RETURN friend_node.name, friend_node.location

  • Gremlin
    
    g = new Neo4jGraph('/path/to/graph/db')
    barbara = g.idx(T,v)[[name:'Barbara']] 
    friends = barbara.out('friend').map
Using Graph Databases

• Use graph databases for…
  • Connected data in link-rich domain (i.e. friends, colleagues, employees, customers, etc.)
  • Routing or dispatch applications with location data (i.e. maps, directions, distances)
  • Recommendation engines (i.e. for products, dating services, etc.)

• Don’t use graph databases for…
  • Applications where many or all data entities need to be updated at once or frequently
  • Data that needs lots of partitioning
Further Matters

- Schema Migrations
- Polyglot Persistence
- SQL, NoSQL, or NewSQL
Schema Migrations

- The structure of data changes regardless of what kind of database it resides in
  - System requirements evolve and the supporting database(s) must keep pace
  - Transition phase – Period of time in which the old and new schema versions must be maintained in parallel

- Challenges
  - Avoid downtime of production database(s)
    - Difficult to do for large systems as DDL to alter structure often requires database object-level locks
  - Ensure database remains usable to all applications during transition phase
    - Different applications will integrate the schema changes at different times
    - Don’t cause errors
    - Don’t corrupt or lose data
  - Minimize transition phase
    - How can all data be migrated as quickly as possible?
    - Does all data need to be migrated?
Schema Changes in Relational Databases

- Challenges specific to RDBMS schema changes
  - Keep database and applications in sync
    - Schema changes applied separately to database and applications
    - Schema changes need to be applied in the correct order
  - Need to ensure that schema changes can be rolled back if there is a problem
  - Schema changes need to be applied to all environments in the same fashion
    - Development, test, staging, production

- Database migration framework can assist with this
  - Logic to execute each schema change is stored in a file which contains a version string
    - Scripts to generate initial database or take a “snapshot” of the current structure of an existing database get the initial version (if the database already exists)
    - May contain logic to upgrade and downgrade the database to/from its version
    - Migration framework is responsible for applying changes up/down to a certain version of the database in the right order
    - Integrated into the project build process so it automatically gets executed in various environments when a new version of the application is introduced there
Database Migration Framework Example

![Database Migration Framework Example](image)

Figure 12.3. New change `007_DiscountedPrice.sql` applied to the database

```
ALTER TABLE orderitem ADD discountedprice NUMBER(18,2) NULL;
UPDATE orderitem SET discountedprice = price;
ALTER TABLE orderitem MODIFY discountedprice NOT NULL;
ALTER TABLE orderitem RENAME COLUMN price TO fullprice;
--//@UNDO
ALTER TABLE orderitem RENAME fullprice TO price;
ALTER TABLE orderitem DROP COLUMN discountedprice;
```
Figure 12.4. DBDeploy upgrading the database with change number 007
Schema Changes in a NoSQL Store

- Implicit schema – the database may be “schema-less”, but the application still must manage the way data is structured

- Incremental migration – read from both schemas and gradually write changes
  - Read methodology:
    - Read the data from the new / updated field(s)
    - If the data is not in the new field(s), read it from the old ones
  - Write methodology:
    - Write data only to the new field(s)
    - Old field may be removed
    - Some data may never be migrated

- Changes to top-level aggregate structures are more difficult
  - Example: make nested order records (inside customers) into top-level aggregates
  - Application must work with both old and new structures
Incremental Migration Example

Figure 12.6. Transition period of schema changes
Polyglot Persistence

- Pick the best tool for the job
  - Different databases are designed specifically for storing and processing different types of data

- Example
  - Many e-commerce sites run entirely on a relational database
  - Alternatively:
    - Keep order processing data in the RDBMS
    - Session and shopping cart data could be separated into a key-value store
      - More transient data which can be copied to RDBMS once an order is placed
    - Customer social data could reside in a graph database
      - Designed specifically to optimize traversing relationships between data
Polyglot Persistence Example

Figure 13.3. Example implementation of polyglot persistence
Web Service Wrappers for Data Stores

- Advantages over direct access to data store
  - Easier and cleaner to integrate the data store with multiple applications
  - Allows database structure to change without needing to update applications that use it
    • Potentially even change the database itself

- Drawbacks
  - Overhead of another layer
  - Sometimes a modified web service actually requires changing applications as well
    • Reduces this likelihood
Web Service Wrapper

Example

Figure 13.5. Using services instead of talking to databases
When to Use NoSQL

• It depends on factors like…

• Programmer productivity (easier to build)
  • When data is mainly collected or displayed in terms of aggregates
  • When the data includes complex, nested, or hierarchical structures
  • When data has a lot of relationships (graph databases)
  • When the data is non-uniform
  • When the database logic can be encapsulated into an isolated section of the project

• Data-access performance (faster)
  • When data needs to be clustered (fragmented and/or replicated)
  • When aggregate data would need to be joined from multiple tables in an RDBMS
  • When complex relational data needs to be queried (graph databases)
When Not to Use NoSQL

- Most of the time
  - Relational databases are well-known, mature, and have lots of tools
- When the need for transactional consistency outweighs performance or productivity concerns
- When many different applications (with different developers/owners) will access the data
- When strong security measures are required at the database level to protect data
NewSQL

• Emerging (bleeding edge) set of databases which promise to let you Have Your Cake and Eat It Too

• Promises the scale and performance of NoSQL with the ACID transactions of relational databases

• Usually supports the relational data model and use SQL

• Flavors
  • New database platforms (i.e. Google Spanner)
  • Engines that integrate with existing relational databases like MySQL
  • Sharding middleware – transparently split (relational) databases across multiple nodes