CSC 560	Modern DBMS Architectures	Alexander Dekhtyar

Course Project Overview
Project Topics

Project Topics

The goal of the course project is to develop, from scratch, a prototype of a non-traditional distributed DBMS architecture.

From scratch. You are responsible for implementing all DBMS-related code. You may not use open source implementations of the algorithms/techniques/solutions even if they come from the code bases of the DBMS you are trying to "replicate".

Prototype. The stress in this class is on **building from scratch**. Therefore, there is no expectation of high performance from the systems that you build. Additionally, due to the length of the quarter, you will be allowed to implement simplified solutions to a variety of tasks, and to make simplifying assumptions about the work of your system and its operating environment. All of this will need to be properly documented, but you are building simplified versions of the systems, aka, prototypes.

Non-traditional. One thing we **will not** do in this class is build a relational DBMS with row-based storage. The majority of DBMS architectures you can choose from (read below) are non-relational DBMS (sometimes also referred to as NoSQL DBMS). However, at least one possible architecture is for relational DBMS using an alternative way storing data. Because of this, rather than saying that we are building NoSQL DBMS, we use the term DBMS with non-traditional architecture.

Distributed. The key reason why non-traditional (and specifically NoSQL) DBMS architectures proliferated is the need to handle more data than a single server can hold and respond to more inquiries than a single server can handle. All your DBMS will have to be distributed. Each team will receive access to 5 virtual machines. While there is no requirement that the final system must deploy on all five, it will have to use at least two. You will select the type of distributed architecture you want to build (Consistent, Partition-tolerant (CP) or Available, Partition-tolerant (AP))[1].

DBMS Architectures to Replicate

While modern non-traditional DBMS have vastly different architectures, they can still be roughly partitioned into a number of groups, with each group combining together systems with similar purpose and internal organization.

Each team shall select one type of archtecture to work on throughout the quarter. The list of archtectures is presented below with short descriptions of each type and some notes about the advantages and disadvantages of choosing each specific type.

DBMS Architecture 1: Distributed Key-Value Stores

Description. Key-value stores are systems devoted to the problem of storing pairs $(id, data)$ where id is a unique key and $data$ is some data record of arbitrary length. These systems usually have a very simple query language/API: with only put() and get() methods that insert a single object and retrieve a single object given its key value. *However*, key-value stores are designed to store massive quantities of key-value pairs and to retrieve them **very fast**.

The key distinguishing feature of a pure key-value store is that there is no way to query the *contents* of the data portion of the record: all record retrieval goes by key (and keys are typically simple).

Examples. Amazon's Dynamo[2] is a typical example of a key-value store (at least, the earlier versions of the system). MongoDB[3] and CouchDB[4] are two other key-value DBMS, although both have some facilities to get "inside" the stored values.

Implementation Advantages. This is probably the most straightforward type of DBMS architecture. The querying functionality is very simple.

Disadvantages. Key-value stores are used for their fast performance and availability of services. Your implementation would need to take it into account.

DBMS Architecture 2: Distributed In-memory DBMS

Description. RAM sizes of up to 32-64Gb and distributed systems mean that unless the datasets that need to be stored and processed are of the hundreds of Terabytes scale, the data can realistically be stored in RAM

[1] More on this and the CAP theorem during the first few weeks of the course.
[2] http://aws.amazon.com/dynamodb/
[3] www.mongodb.org
[4] http://couchdb.apache.org/

on a distributed system. This gives rise of a wide range of possible DBMS architectures that rely on in-memory processign and never use secondary storage during query processing tasks[5]

The in-memory DBMS is probably the most diverse DBMS architecture category[6] On one end of the scale distributed caches/distributed in-memory key-value systems (e.g., memcached) can be viewed as in-memory DBMS. On the other side are in-memory versions of relational DBMS architectures.

Examples. memcached[7] is an example of a distributed cache. **VoltDB**[8] is a high-performance in memory relational (NewSQL) DBMS. Aerospike[9] is an example of an in-memory NoSQL DBMS, which is essentially a key-value store, but with a wider range of querying options available.

Advantages. When working on an in-memory DBMS you will have more freedom of choice with respect to the specific design than when working on other types of architectures. In memory distributed key-value stores are on the simpler scale of DBMS architectures. But systems like VoltDB are much more complex.

Disadvantages. This category has not just the widest variety but also hides the largest range of difficulty for projects. You need to pay extra care when selecting your design.

DBMS Architecture 3: Non-relational Columnar DBMS

Description. These DBMS were built primarily to support distributed MapReduce processing. The common features among all DBMS of this type are the three-level nomenclature: a record is a collection of column values organized into column families. Values can be present or absent, the DBMS try to store efficiently only the data that is present in the system. Querying Columnar DBMS is usually done via MapReduce jobs, although some DBMS have native APIs that retrieve data from "inside" the records.

Examples. Google's BigTable[10] and its open-source variant, Apache's HBase[11] were the starting point of this architecture, as Google was looking to put a DBMS-style back-end to its MapReduce processing. Facebook-developed Apache Cassandra[12] is another popular column DBMS.

Advantages. One of the most interesting/creative designs to implement. Relatively straightforward in a sense that most column stores have similar architecture.

Disadvantages. A fairly complex organization and relatively little room

[5] Some in-memory DBMS may use secondary storage for archival purposes.

[6] And therefore, I will allow multiple teams to pick it, as long as the actual projects wind up being different.

[7] http://memcached.org/

[8] http://voltdb.com/

[9] http://www.aerospike.com/

[10] http://research.google.com/archive/bigtable.html

[11] http://hbase.apache.org/

[12] http://cassandra.apache.org/

for creativity.

DBMS Architecture 4: Relational Column-oriented DBMS

Description. Not to be confused with NoSQL column stores like BigTable, the DBMS belonging to this category are relational DBMS that choose to store data in a different way. Unlike traditional relational DBMS that store data in row format (the data for each record is stored together in the same location), the column-based relational DBMS store data by columns: all data from a single attribute of a table is stored in the same location. This simple change in underlying storage architecture has drastic effects on how the rest of the DBMS is organized, and thus, is of interest to us.

Examples. The key academic example of a colum-based DBMS is MonetDB[13] (there are extensions of MonetDB to other DBMS architectures, e.g., it was the first successful fully XQuery-compliant XML DBMS, but the core system is a column-based relational DBMS). InfiniDB[14] is a commercial DBMS with a full SQL service available.

Advantages. One of the most clear architectures to implement - the lower levels of the system you build are probably the most straighforward ones (even when factoring in for the distributed system aspect).

Disadvantages. These are relational DBMS, so the query language is SQL, which is huge, requires parsing, query compilation and optimization. Your implementation will either need to do this for a very simple subset of SQL, or you will have to stop at providing the relational algebra API layer.

DBMS Architecture 5: Semistructured DBMS

Description. A lot of data around is is not easily put into a collection of flat tuples. DBMS working with XML data have been out there since late 1990s. As of recent, JSON format supplanted XML as the means of expressing schema-less semistructured data. Otherwise known as document-oriented DBMS, the database management systems in this category index individual semistructured documents (usually provided in JSON or XML format) and implement powerful query languages that allow access to the contents of each document.

Examples. This is probably the most popular type of NoSQL DBMS, as most pure key-value DBMS are trying to make their products more flexible and applcable to a wider range of use cases. MarkLogic[15] is an example of a commercial XML DBMS that was a document store from the get go. MongoDB[16], and CouchBase[17] are examples of DBMS that evolved from

[13] http://www.monetdb.com/
[14] http://www.infinidb.co/
[15] http://www.marklogic.com
[16] http://www.mongodb.org
[17] http://www.couchbase.com/

key-value stores to documents stores over time.

Advantages. Probably the most "mainstream" of all NoSQL DBMS architectures, and therefore, the most useful to have implemented.

Disadvantages. Will require a non-trivial subset of XPath/XQuery implemented as a query language (even if you are implementing a JSON document store, the query language still has to posess the ability to express paths withing the document, so it'd have to be equivalent to a non-trivial subset of XPath).

DBMS Architecture 6: Graph DBMS

Description. In a lot of situation, the dataset that needs to be processed comes as a set of (subject, object, property) triples. In such cases, as well as in cases where the data is an actual representation of a very large graph, graph databases are an appropriate solution.

Examples. The most well-known graph database is Neo4j[18].

Advantages. A really interesting use case to work on.

Disadvantages. High complexity of implementations.

[18] http://www.neo4j.org/