Cogset vs. Hadoop
Measurements and Analysis

Steffen Viken Valvåg
Dag Johansen Åge Kvalnes

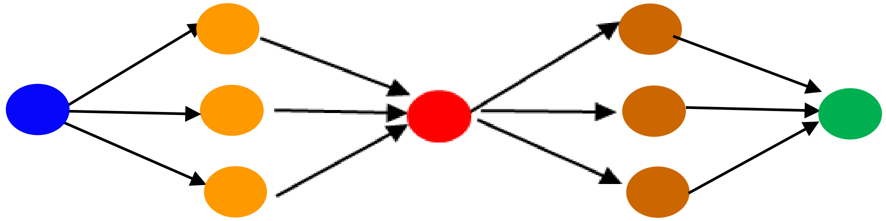
Dec 1, 2010
MAPRED2010

Context and Background

- Part of the international research project **iAD**, focusing on *information access* applications
- Hosted by the Norwegian search company FAST (now a Microsoft subsidiary) in collaboration with:
  - Cornell University (Cornell), Dublin City University (DCU), BI Norwegian School of Management (BI), Norwegian University of Science and Technology (NTNU), University of Oslo (UiO), **University of Tromsø (UiT),** Accenture
- Broad range of research topics, including run-times to facilitate distributed data processing (analytics) in cloud environments.

Analytics (large scale data processing)

- **Analytics** are important for information access applications
  - Constructing indexes, analysing trends, sentiments, and link structures, mining and correlating logs, recommending items, etc.
  - Data-intensive computations that must be distributed for efficiency
- **Run-times** automate *scheduling* of processes on cluster machines, *monitor* progress, ensure *fault tolerance*, and support efficient *data transfer* between processes.



- Widely adopted framework (and programming model): **MapReduce**
  - Hadoop is the most widely deployed open source implementation

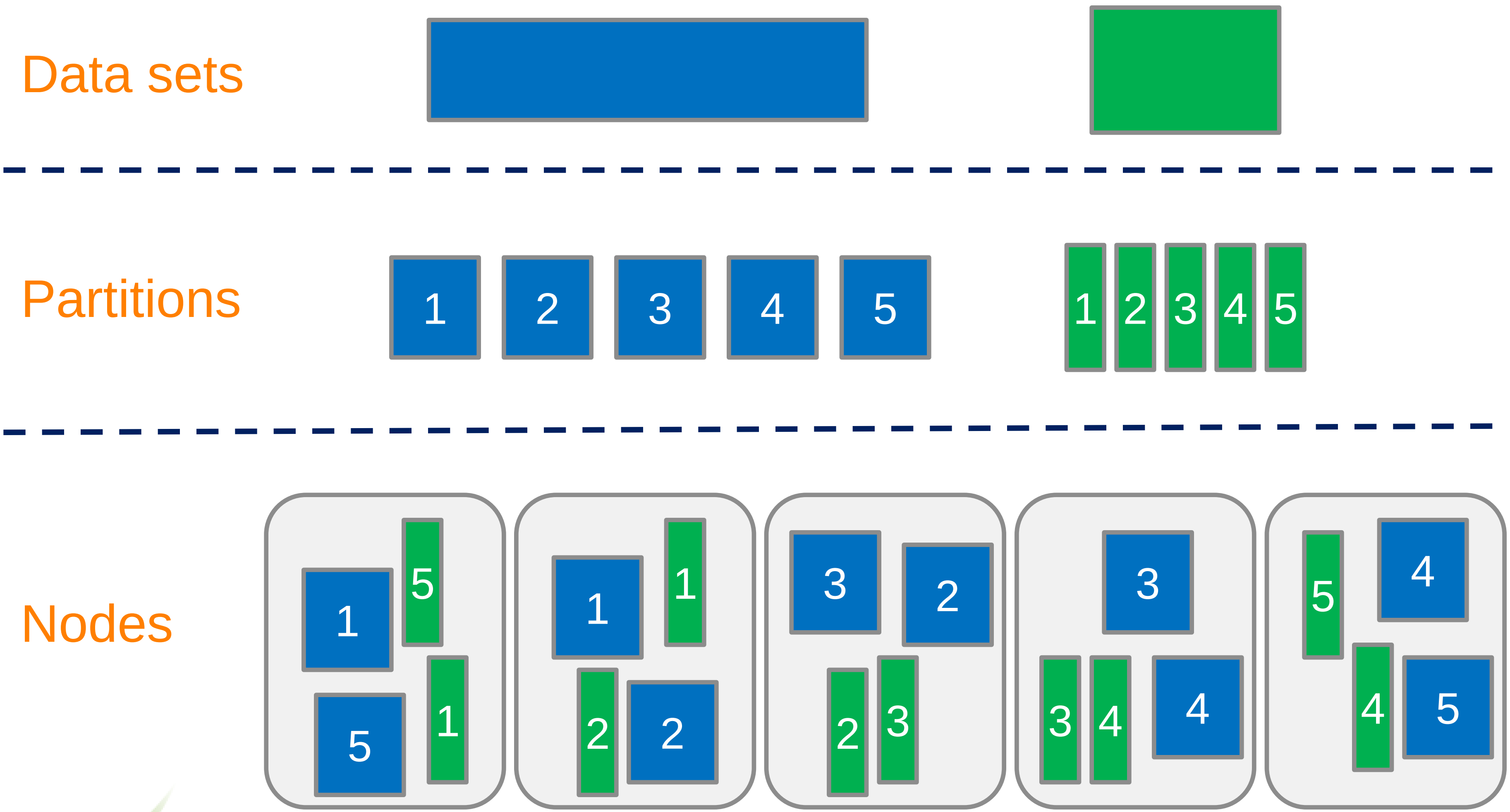
Cogset

- A generic engine for:
  - Reliable storage of data
  - Parallel processing of data
- Inspired by both MapReduce and databases
  - Schema-free data model
  - Data processing with user-defined functions
  - Push-based static routing of records
  - Novel mechanisms for fault tolerance and load balancing
- Supports several high-level programming interfaces
  - Oivos (NPC2009): Declarative workflows
  - Update Maps (NAS2009): Key/value interface
  - **MapReduce:** Compatible with Hadoop

Overview of Cogset

– Data sets are stored as a number of partitions
  - Distributed and replicated for redundancy
– Data is accessed by performing *traversals*
  - Functional interface, specifying a user-defined visitor function (UDF) to be invoked in parallel for all partitions.
  - Visitors may read multiple data sets and add records to multiple new or existing data sets.
  - Output is atomically committed once a traversal completes.

Partitioning



Traversals

- High-level algorithm:
  - For each partition, select a node that is hosting the partition and evaluate the visitor function there
  - Collect output records from the visitor and route them to the appropriate nodes
  - Once all partitions are processed, commit all output
- Implementation:
  - Fully distributed scheduling algorithm
  - Each node monitors and coordinates with its “neighbors”, which are nodes with replicas in common
  - Slow nodes are detected and off-loaded by their neighbors
  - Status and progress is reported to the client, which acts as the “master” for a given traversal

Data placement and locality

- Motivation for integrating storage and processing
  - When storage is decoupled from processing, data locality is harder to ensure
- Clients may influence data locality by choosing how to partition data
  - Corresponding partitions of different data sets are always co-located, and accessible together by a visitor function
- Example: Hash join
  - With Cogset, a hash join can be implemented by a single traversal without repartitioning data
  - With traditional MapReduce, a hash join must first repartition all data in the Map phase

MapReduce support in Cogset

– Highly compatible with Hadoop
  - Construct a JobConf object in the regular way, then run the job using Cogset rathen than Hadoop.
  - New-style interfaces (Hadoop 0.19+) also supported
– Implemented as two traversals
  - The first traversal implements the map phase, using a visitor that reads all input records and passes them to the user-defined Mapper (and Combiner).
  - The second traversal implements the reduce phase, sorting each partition and applying the Reducer.

The MR/DB benchmark

- Developed by Pavlo et al. for the SIGMOD 2009 paper "A comparison of approaches to large-scale data analysis"
- Designed to compare the performance of MapReduce and Parallel Databases.
  - Originally used to compare Hadoop, Vertica, and a second parallel database system (DB-X).
  - Subsequently used to evaluate HadoopDB in a separate paper.
  - Features 5 tasks that may be expressed either as SQL queries or as MapReduce jobs, with provided source code for Hadoop.
- We used MR/DB to compare the performance of Cogset, when employed as a MapReduce engine, to Hadoop.
  - The exact same MapReduce benchmark code was executed using both Hadoop and Cogset

MR/DB benchmark tasks

– Grep: Sequential scan of a data set

- 1 map-only job

```
CREATE TABLE Data (key VARCHAR(10) PRIMARY KEY, field VARCHAR(90));

SELECT * FROM Data WHERE field LIKE '%XYZ%';
```

– Select: Sequential scan, less selective

- 1 map-only job

```
CREATE TABLE Rankings (pageURL VARCHAR(100) PRIMARY KEY,
                       pageRank INT,
                       avgDuration INT);

SELECT pageURL, pageRank FROM Rankings WHERE pageRank > X;
```

MR/DB benchmark tasks

– Aggregate: Aggregate total revenue per IP
  - 1 full MapReduce job

```
CREATE TABLE UserVisits (sourceIP VARCHAR(16),
                         destURL VARCHAR(100),
                         visitDate DATE,
                         adRevenue FLOAT,
                         userAgent VARCHAR(64),
                         countryCode VARCHAR(3),
                         languageCode VARCHAR(6),
                         searchWord VARCHAR(32),
                         duration INT );

SELECT sourceIP, SUM(adRevenue)FROM UserVisits GROUP BY sourceIP;
```

MR/DB benchmark tasks

– Join: Complex two-way join + aggregation

- 3 MapReduce jobs

```
SELECT INTO Temp sourceIP,
                 AVG(pageRank) as avgPageRank,
                 SUM(adRevenue) as totalRevenue
  FROM Rankings AS R, UserVisits AS UV
 WHERE R.pageURL = UV.destURL
   AND UV.visitDate BETWEEN Date('2000-01-15')
   AND Date('2000-01-22')
 GROUP BY UV.sourceIP;

SELECT sourceIP, totalRevenue, avgPageRank
  FROM Temp
 ORDER BY totalRevenue DESC LIMIT 1;
```

MR/DB benchmark tasks

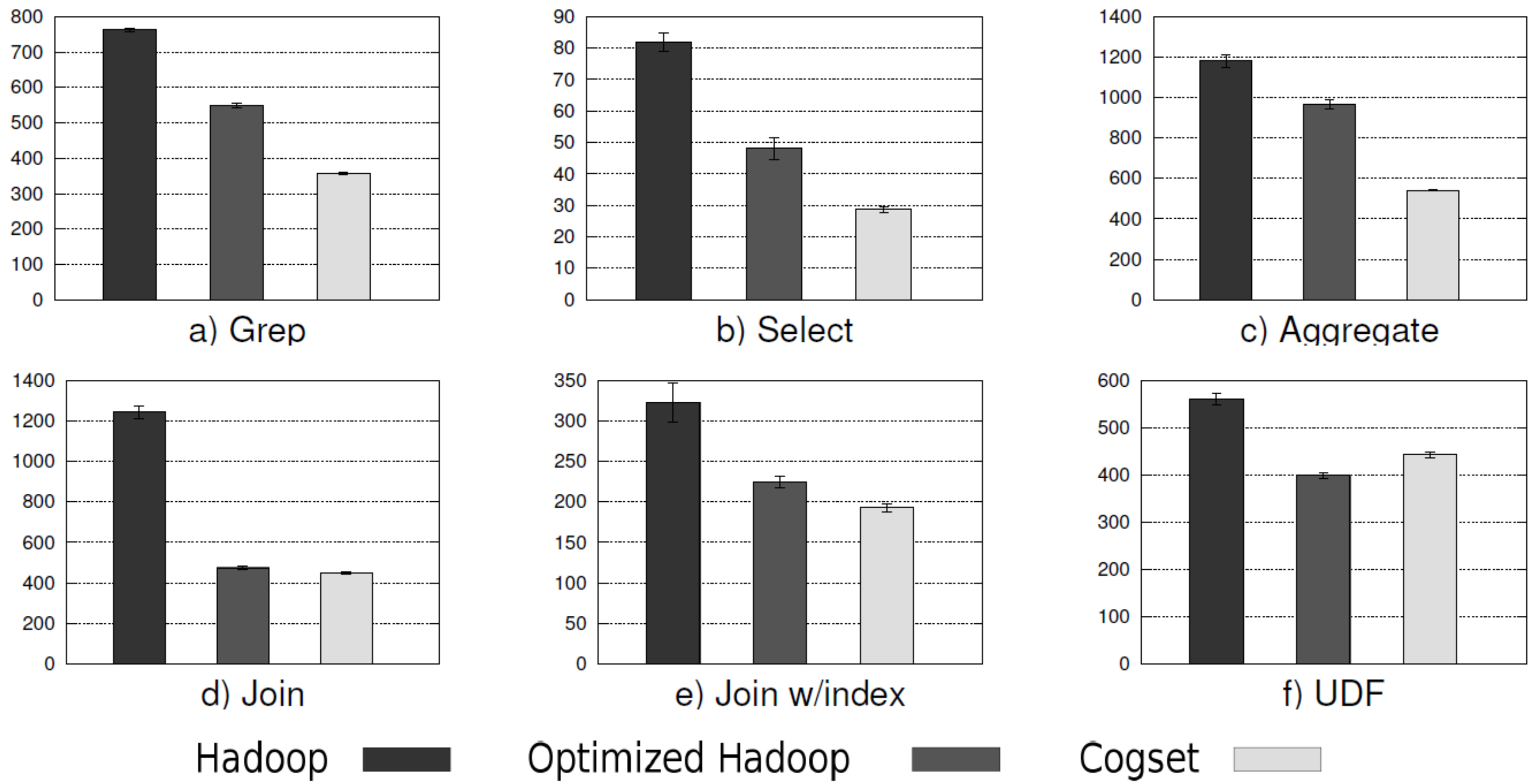
– UDF: Parse hyperlinks from a set of HTML documents and invert the link graph
  - 1 MapReduce job

```
CREATE TABLE Documents (url VARCHAR(100) PRIMARY KEY,
                        contents TEXT );

SELECT INTO Temp F(contents) FROM Documents;
SELECT url, SUM(value) FROM Temp GROUP BY url;
```

  - F is a user-defined function that must be integrated into the query plan by the parallel databases

MR/DB results for 25 nodes



- Cogset improves performance significantly for several benchmark tasks
- When investigating, we also discovered ways to improve Hadoop's benchmark performance by making various optimizations (these results are labeled "Optimized Hadoop")

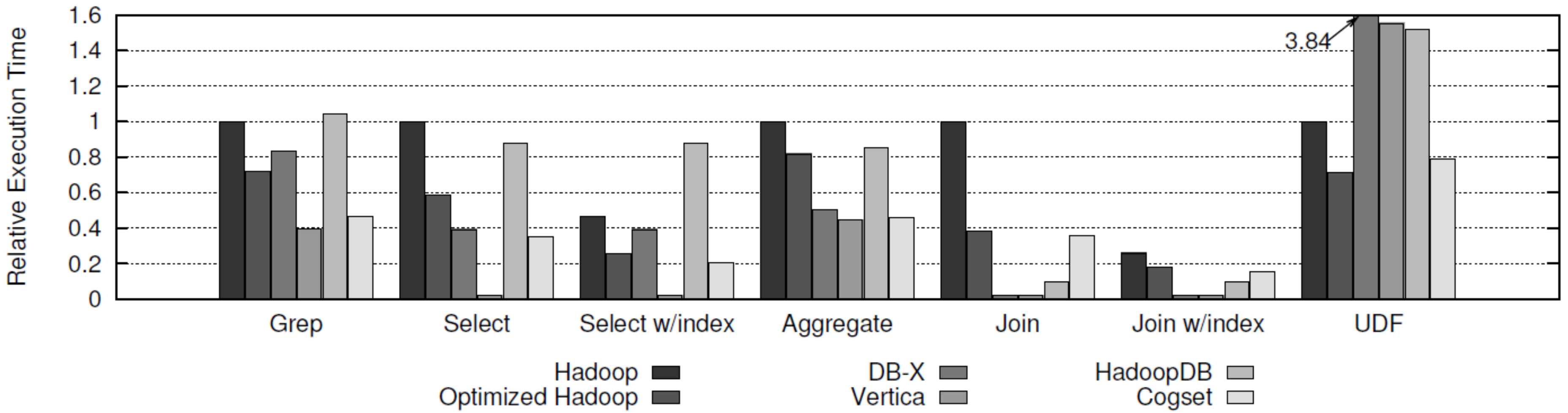
Hadoop bottleneck: Task scheduling

- Hadoop's task trackers communicate with the central job tracker using heartbeat RPCs
  - Heartbeats occur at most every 3 seconds, and task completion is only reported then
  - Consequently, **task trackers may go idle** if tasks are short-lived
- Unexpected interaction with HDFS block size
  - Bigger block size = more work per mapper = less idle time
- For Grep, task trackers were idle **34%** of the time using the default Hadoop configuration
  - A simple patch allowed us to report completed tasks immediately
- Hadoop 0.21 introduced a new option that may help
  - mapreduce.tasktracker.outofband.heartbeat
  - Enable this to send out-of-band heartbeats upon task completion

Hadoop bottleneck: Multi-core CPU utilization

- For sequential scanning of data, and whenever costly UDFs are invoked, Hadoop quickly becomes CPU bound
  - Multiple cores are not well utilized, so there may well be spare CPU cycles that go unused
  - Increasing the number of concurrent processes is ineffective, because of memory footprint and less optimal I/O access patterns
- Cogset employs multiple threads to read, parse and process records in parallel
  - Fully exploits all cores when costly UDFs are employed
- By implementing a similar approach in Hadoop, plugged in as a custom input format, performance was greatly improved

Relative performance to other systems



– When comparing the relative performance to Hadoop, Cogset matches the performance of previously benchmarked parallel database systems
  - Index structures skew some results in favor of the parallel database systems
  - For sequential scanning and aggregation, Cogset matches or outperforms Vertica and DB-X.

Conclusion

- The MR/DB benchmark primarily exposed implementation weaknesses in Hadoop; the results are not due to fundamental limitations of the MapReduce programming model
  - Cogset matches the performance of parallel databases while supporting the MapReduce programming model
- Previous criticism of the MR/DB benchmark has pointed out that the UDFs and record formats employed are inefficient
  - Cogset tolerates costly UDFs using multi-threading
  - This closes much of the performance gap to parallel databases
  - Similar improvements are possible with Hadoop, but may require some restructuring
- Hadoop's task scheduling is prone to leaving nodes idle
  - Serious problem that affects both throughput and latency
  - Straightforward to fix

Questions?

How Cogset improves performance

- Direct routing of data between computing nodes
  - Avoids temporary storage of transient data
  - Entails novel approaches to fault tolerance and load balancing
- Visitor-based data processing integrated into the storage layer
  - Transfer the *code* to the *data* to reduce bandwidth consumption
- Fully distributed scheduling and monitoring algorithm
  - Monitors peers with common data replicas and dynamically balances load
- Multi-threaded program structure to exploit all available CPU capacity
  - Essential for good performance on multi-core architectures

- Experience with Cogset also used to *improve Hadoop* in several ways
  - Inefficient scheduling algorithm identified and improved
  - Performance critical Cogset code refactored into Hadoop plugins
  - CPU hotspots reduced using multi-threaded code on the critical path