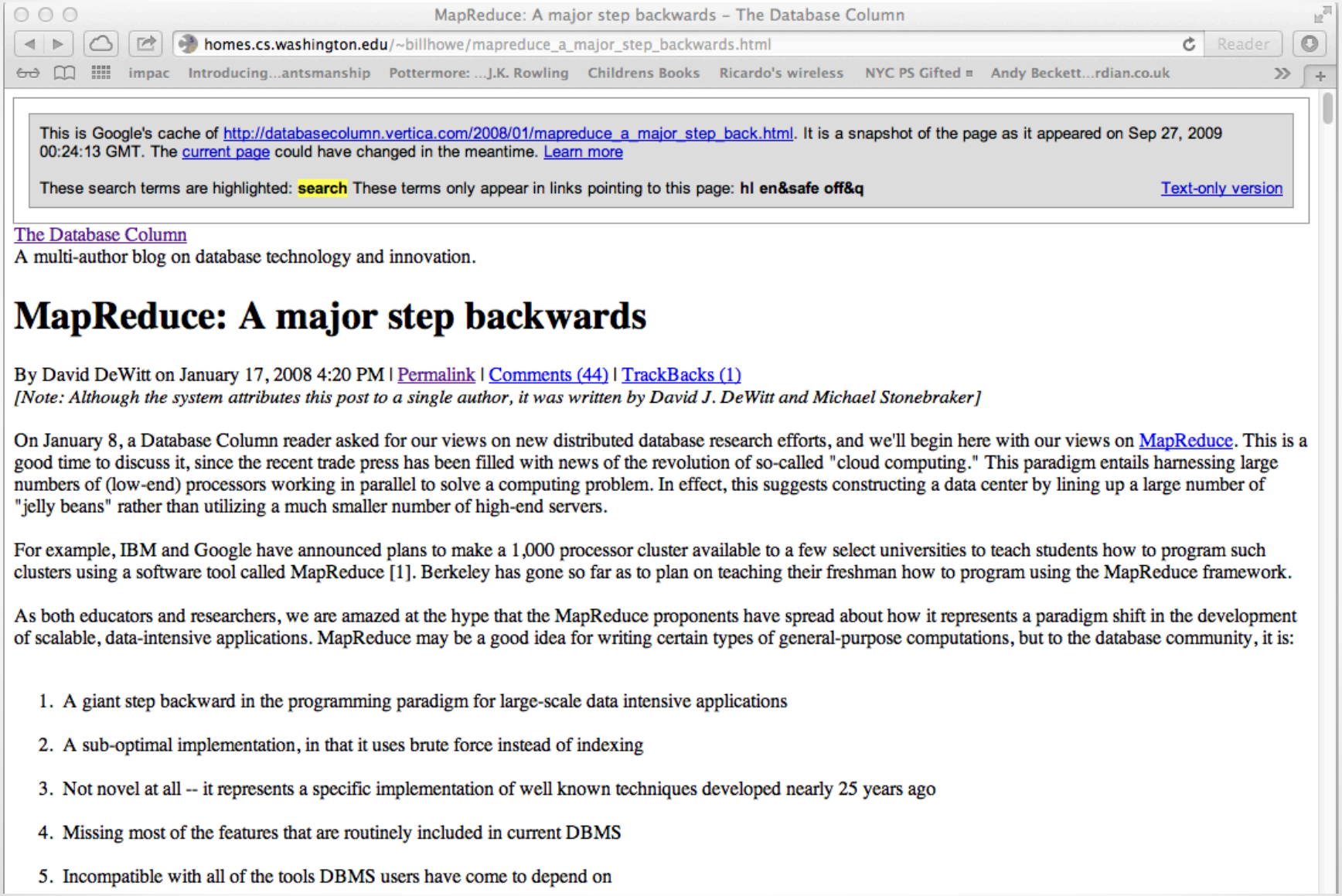
Hadoop
vs.
Parallel Databases

Juliana Freire

The Debate Starts…



The Debate Continues…

- A comparison of approaches to large-scale data analysis. Pavlo et al., SIGMOD 2009
  - *Parallel DBMS beats MapReduce by a lot!*
  - Many were outraged by the comparison
- MapReduce: A Flexible Data Processing Tool. Dean and Ghemawat, CACM 2010
  - Pointed out inconsistencies and mistakes in the comparison
- MapReduce and Parallel DBMSs: Friends or Foes? Stonebraker et al., CACM 2010
  - Toned down claims…

Outline

- DB 101 - Review
- Background on Parallel Databases – for more detail, see Chapter 21 of Silberschatz et al., Database Systems Concepts, Fifth Edition
- Case for Parallel Databases
- Case for MapReduce
- Voice your opinion!

Storing Data: Database vs. File System

- Once upon a time database applications were built on top of file systems…
- But this has many drawbacks:
  - Data redundancy, inconsistency and isolation
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task, e.g., search people by zip code or last name; update telephone number
  - Integrity problems
    - Integrity constraints (e.g., num_residence = 1) become part of program code -- hard to add new constraints or change existing ones
- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out, e.g., John and Mary get married, add new residence, update John's entry, and database crashes while Mary's entry is being updated…

Why use Database Systems?

- Declarative query languages
- Data independence
- Efficient access through optimization
- Data integrity and security
  - Safeguarding data from failures and malicious access
- Concurrent access
- Reduced application development time
- Uniform data administration

Query Languages

- *Query languages:* Allow *manipulation* and *retrieval* of data from a database
- Queries are posed wrt data model
  - Operations over objects defined in data model
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic
  - Allows for *automatic* optimization

SQL and Relational Algebra

- Manipulate sets of tuples
- σ_c R= select -- produces a new relation with the subset of the tuples in R that match the condition *C*
  - $\sigma_{\text{Type = "savings"}}$ Account
  - SELECT * FROM Account
    WHERE Account.type = 'savings'
- $\pi_{\text{AttributeList}}$ R = project -- deletes attributes that are not in *projection list*.
  - $\pi_{\text{Number, Owner, Type}}$ Account
  - SELECT number, owner, type FROM Account

SQL and Relational Algebra

- Set operations: Union intersection
- A X B : cross-product—produce every possible combination of tuples in A and B
  - Teaches X Course
  - SELECT * FROM Teacher, Course
- A $\bowtie_{condition}$ B: joins tables based on condition
  - Teacher $\bowtie_{Teacher.c\_id = Course.id}$ Course
  - SELECT * FROM Teacher, Course
    WHERE Teacher.c_id = Course.id

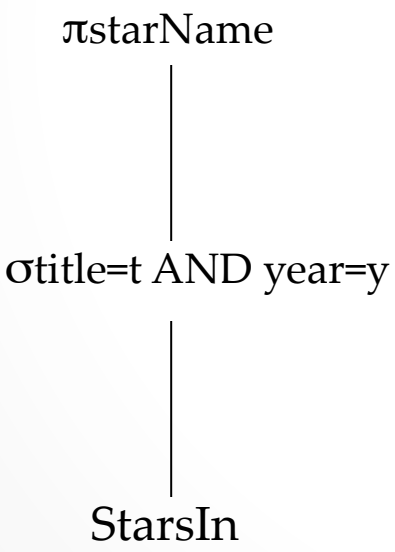
Query Optimization and Evaluation

- DBMS must provide efficient access to data
- Declarative queries are translated into imperative query plans
  - Declarative queries → logical data model
  - Imperative plans → physical structure
- Relational optimizers aim to find the best imperative plans (i.e., shortest execution time)
  - In practice they avoid the worst plans…

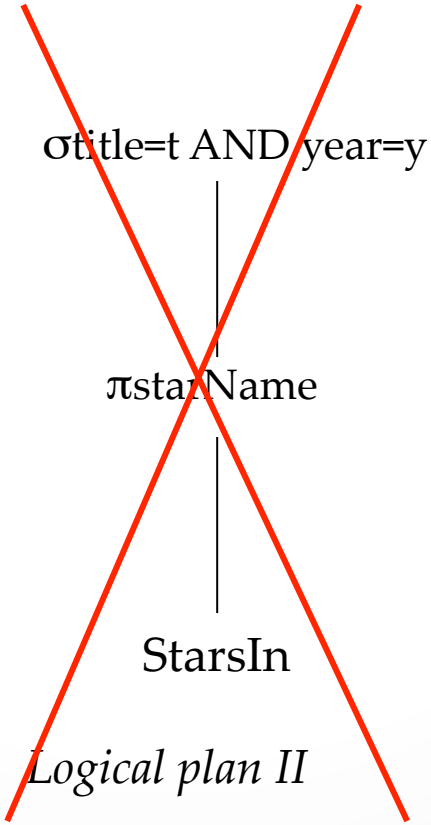
Example: Logical Plan

StarsIn(title, year, starName)

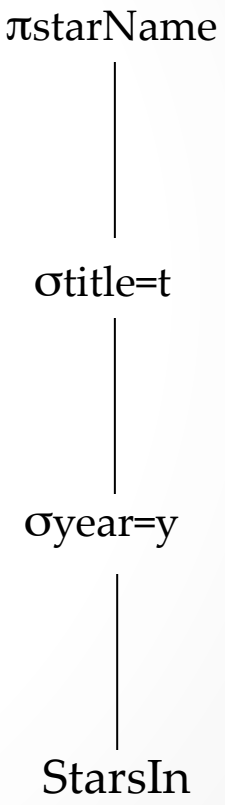
```
SELECT starName
FROM StarsIn
WHERE title = t AND year = y;
```



Logical plan I



Logical plan II

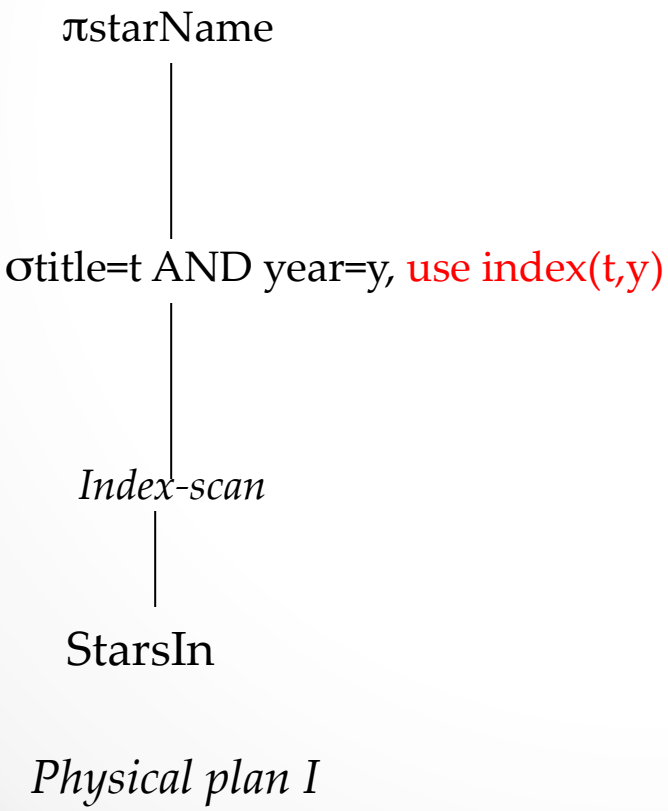


Logical plan III

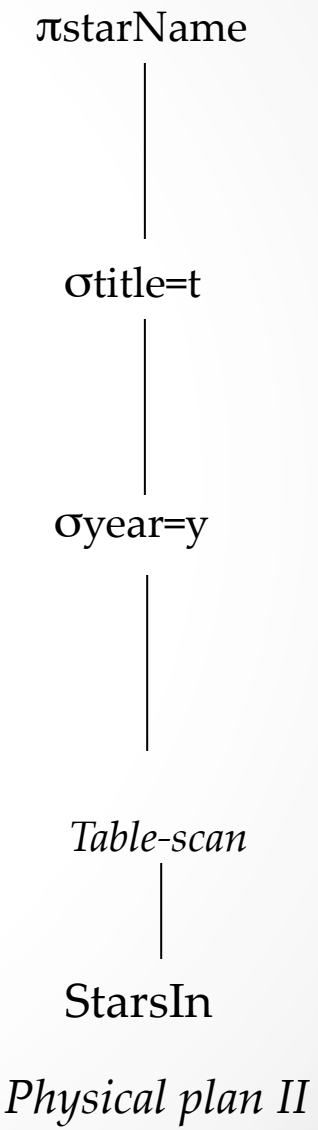
Example: Physical Plan

StarsIn(title, year, starName)

```
SELECT starName
FROM StarsIn
WHERE title = t AND year = y;
```



Physical plan I

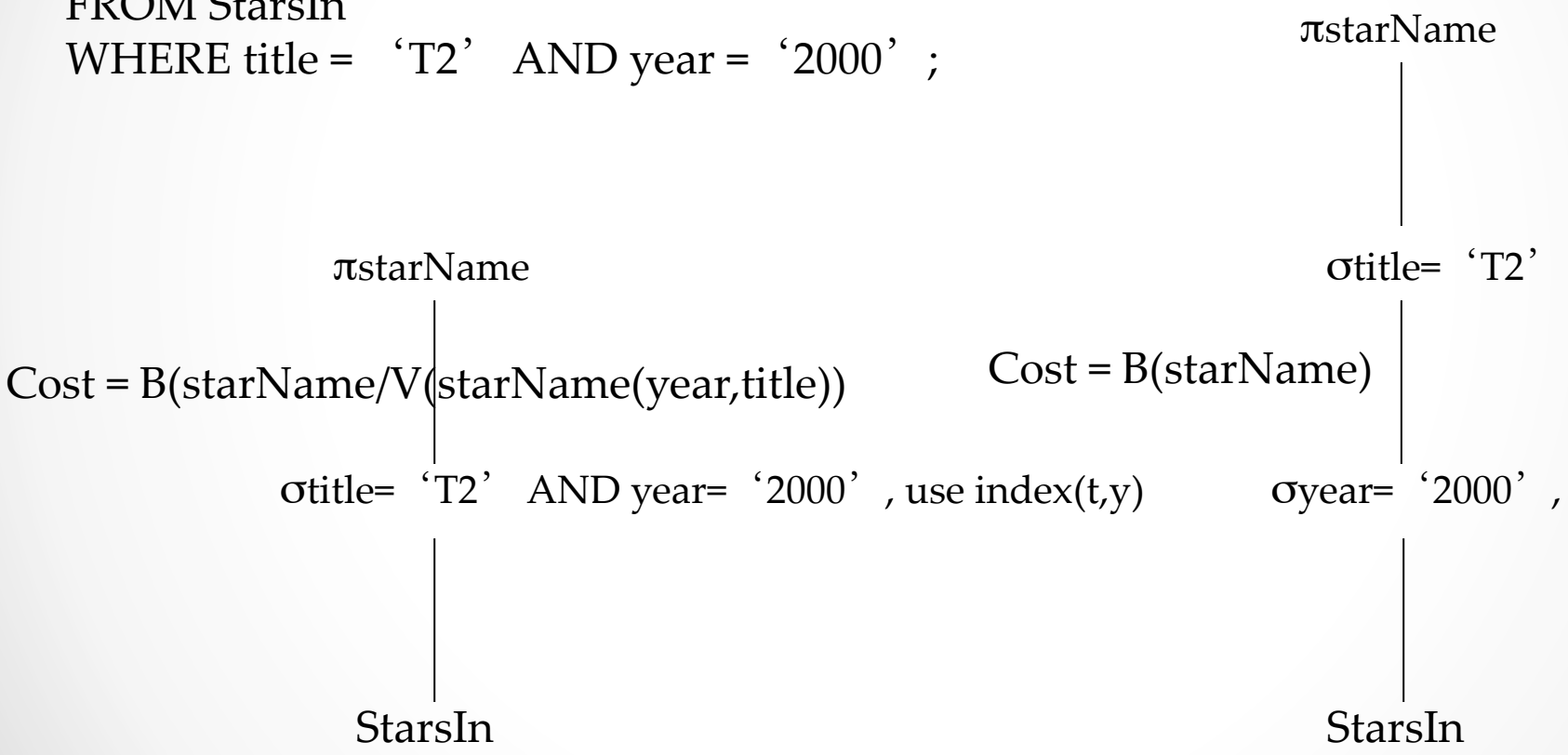


Physical plan II

Example: Select Best Plan

StarsIn(title, year, starName)

```
SELECT starName
FROM StarsIn
WHERE title = 'T2' AND year = '2000';
```



Query Languages

- *Query languages:* Allow *manipulation* and *retrieval* of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logi
  - Allows for much optimization.
- Query Languages != programming lan
  - QLs not expected to be "Turing comple
  - QLs support easy, efficient access to large data sets.
  - *QLs not intended to be used for complex calculations.*

a language that can compute anything that can be computed

Data Independence

- Applications insulated from how data is structured and stored
- *Physical data independence:* Protection from changes in *physical* structure of data
  - Changes in the physical layout of the data or in the indexes used do not affect the logical relations

Title	Year	starName
Argo	2012	Afleck
Batman vs. Superman	2015	Afleck
Terminator	1984	Schwarzenegger
Wolverine	2013	Jackman

Year	Title	starName
2012	Argo	Afleck
2015	Batman vs. Superman	Afleck
1984	Terminator	Schwarzenegger
2013	Wolverine	Jackman

One of the most important benefits of using a DBMS!

Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Integrity Constraints (ICs)

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid) )
```

For a given student and course, there is a single grade.

insert into *enrolled* **values** (1,'cs2345',B)

insert into *enrolled* **values** (1,'cs9223',B)

sid	cid	grade
1	cs9223	A
2	cs2345	B
1	cs2345	B

Integrity Constraints (ICs)

```
CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20),  grade CHAR(2),
    PRIMARY KEY  (sid,cid),
    FOREIGN KEY (sid) REFERENCES Students(sid)
```

Only students listed in the Students relation should be allowed to enroll for courses.

insert into *Enrolled* **values** (1,'cs2345',B)

insert into *Enrolled* **values** (3,'cs2345',B)

sid	cid	grade
1	cs9223	A
2	cs2345	B
1	cs2345	B

delete from *Students*
where sid=1

sid	name
1	john
2	mary

Concurrency Control

- Concurrent execution of user programs is essential for good DBMS performance
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- Interleaving actions of different user programs can lead to inconsistency
  - e.g., nurse and doctor can simultaneously edit a patient record
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system

When should you not use a database?

• • •

Parallel and Distributed Databases

- Parallel database system:
  – Improve performance through parallel implementation

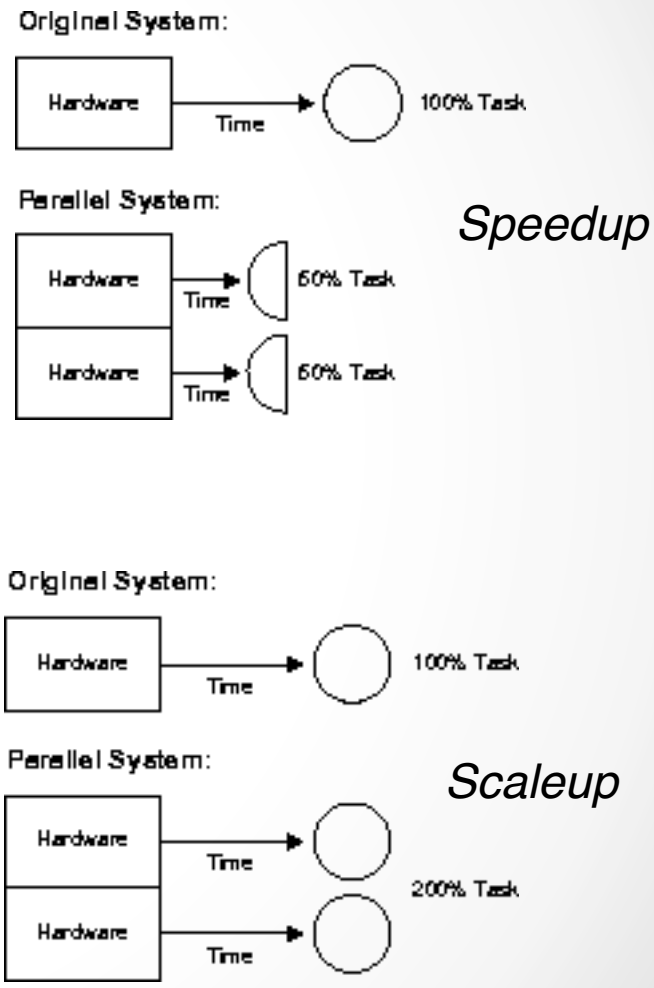
- Distributed database system:

  – Data is stored across several sites, each site managed by a DBMS capable of running independently

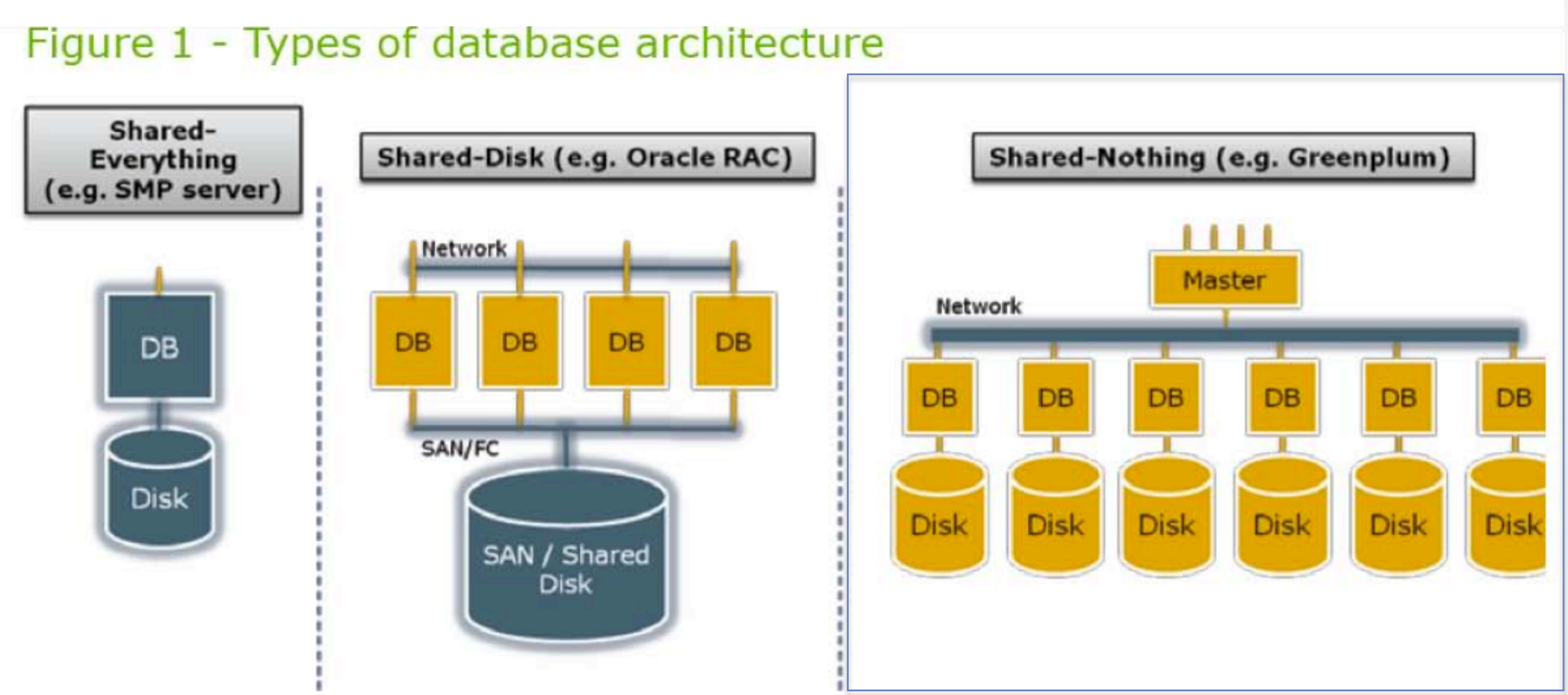
Parallel DBMSs

- Old and mature technology --- late 80's: Gamma, Grace
- Aim to improve performance by executing operations in parallel
- Benefit: easy and cheap to scale
  - Add more nodes, get higher speed
  - Reduce the time to run queries
  - Higher transaction rates
  - Ability to process more data
- Challenge: minimize overheads and efficiently deal with contention

docs.oracle.com

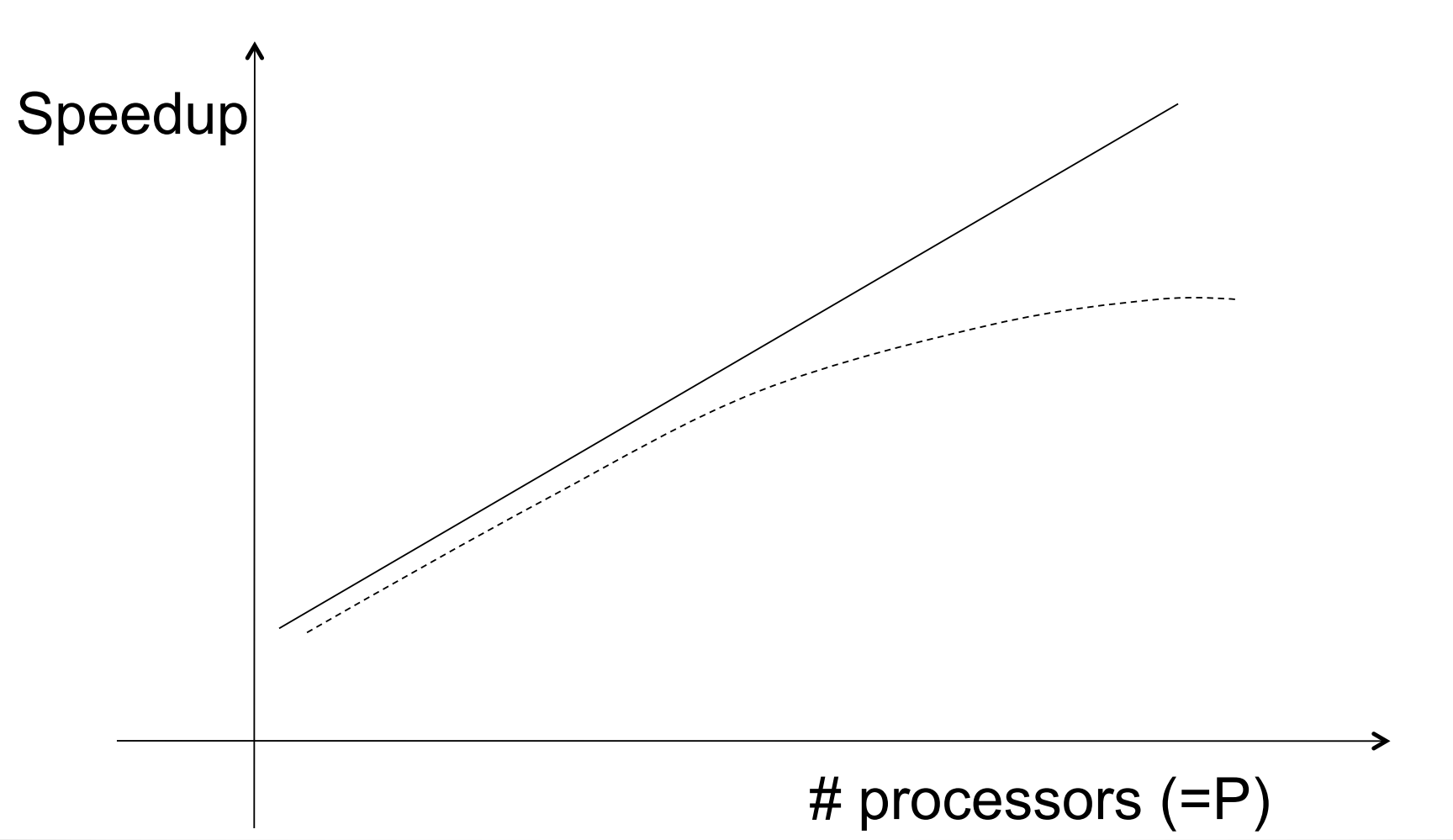


Different Architectures



From: Greenplum Database Whitepaper

Linear vs. Non-Linear Speedup



Achieving Linear Speedup: Challenges

- Start-up cost: starting an operation on many processors
- Contention for resources between processors
- Data transfer
- Slowest processor becomes the bottleneck
- Data skew → load balancing
- Shared-nothing:
  - Most scalable architecture
  - Minimizes resource sharing across processors
  - Can use commodity hardware
  - Hard to program and manage

Does this ring a bell?

What to Parallelize?

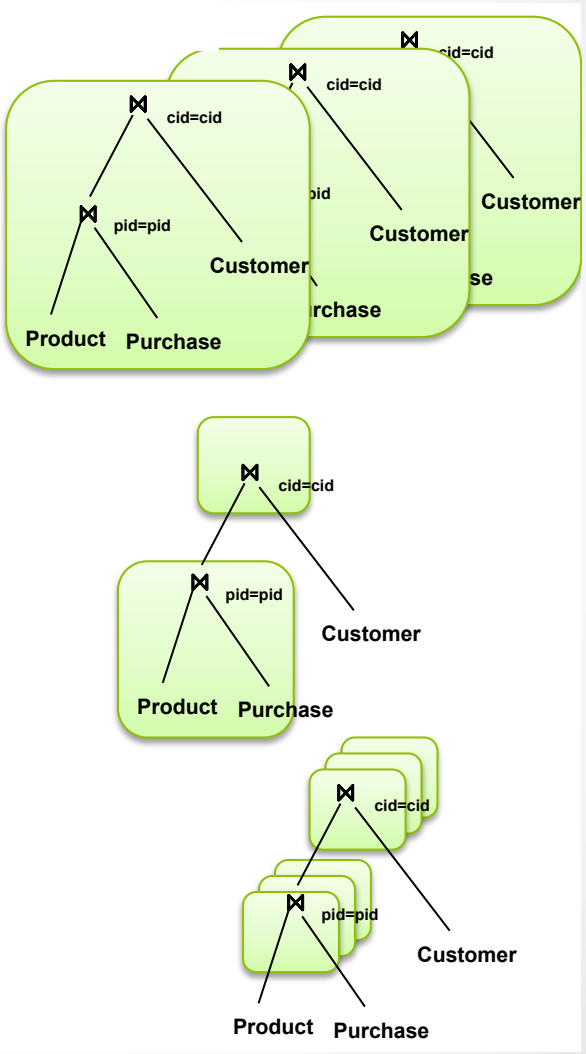
- Inter-query parallelism
  – Each query runs on one processor

- Inter-operator parallelism
  – A query runs on multiple processors

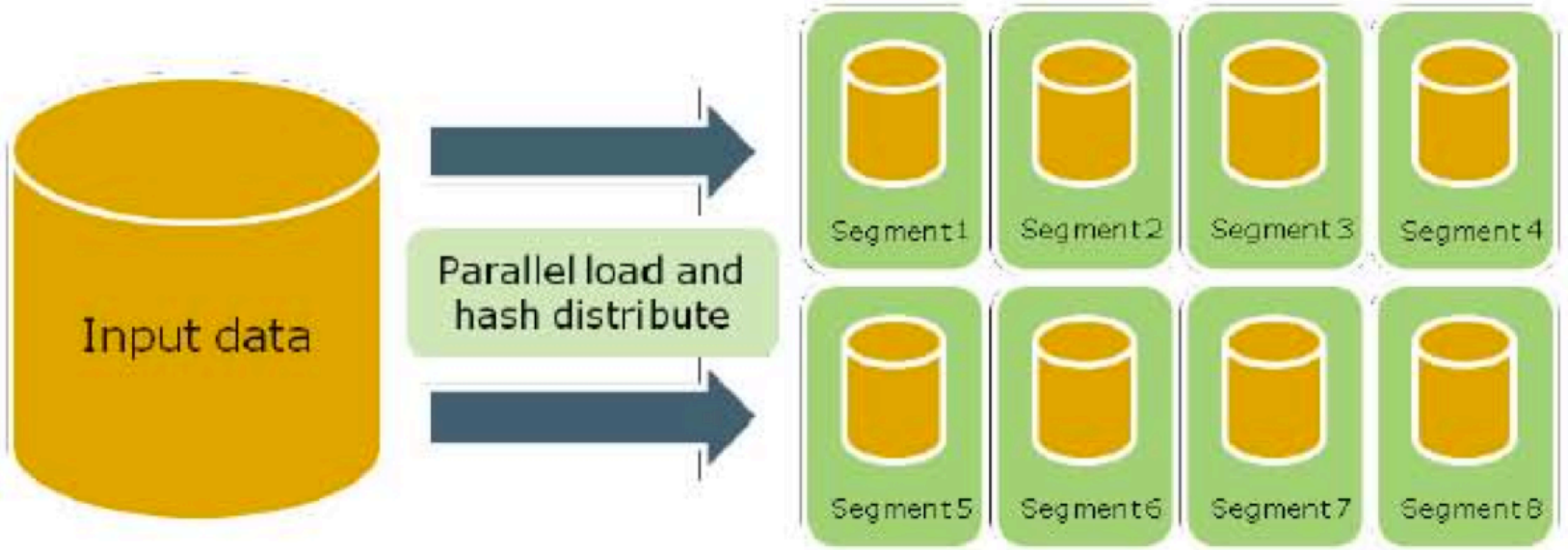
an operator runs on one processor

- Intra-operator parallelism
  – An operator runs on multiple

processors



Query Evaluation



From: Greenplum Database Whitepaper

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk
- Large relations are preferably partitioned across all the available disks
- If a relation consists of *m* disk blocks and there are *n* disks available in the system, then the relation should be allocated **min**(*m,n*) disks
- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples
- Ideal: partitioning should be balanced

Horizontal Data Partitioning

- Relation R split into P chunks R_0, ..., R_{P-1}, stored at the P nodes
- *Round robin:* tuple T_i to chunk (i mod P)
- *Hash based partitioning on attribute A*:
  - Tuple t to chunk h(t.A) mod P
- *Range based partitioning on attribute A*:
  - Tuple t to chunk i if $v_{i-1} < t.A < v_i$
  - E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk 2.

Partitioning in Oracle:
http://docs.oracle.com/cd/B10501_01/server.920/a96524/c12parti.htm
Partitioning in DB2:
http://www.ibm.com/developerworks/data/library/techarticle/dm-0605ahuja2/

Range Partitioning in DBMSs

```
CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY RANGE(sales_date)
(
PARTITION sales_jan2000 VALUES LESS
THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS
THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS
THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS
THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```

```
CREATE TABLE sales_hash
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
week_no       NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4
STORE IN (data1, data2, data3,
data4);
```

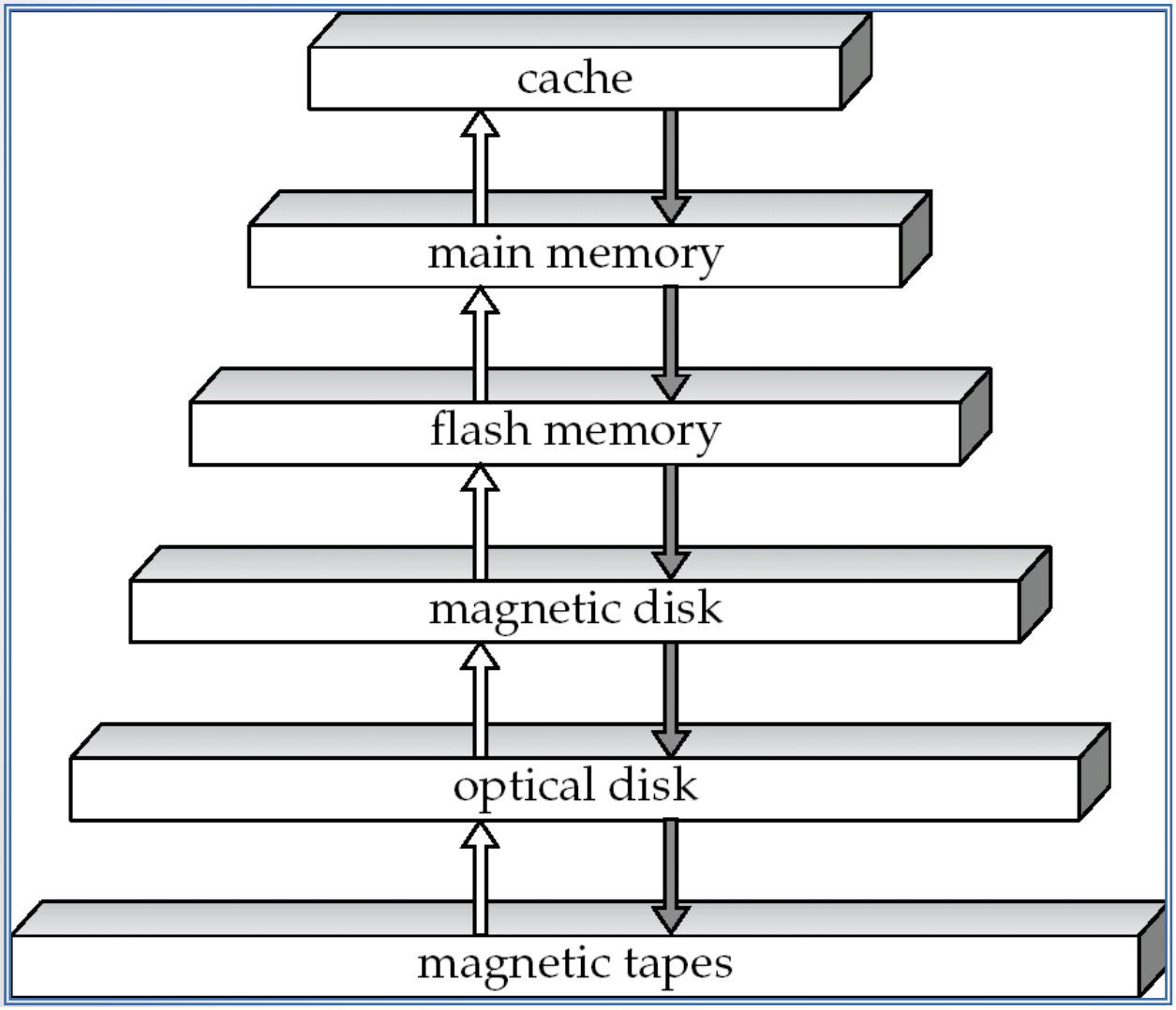
Databases 101

Why not just write customized applications that use the file system?

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task, e.g., search people by zip code or last name; update telephone number
- Integrity problems
  - Integrity constraints (e.g., age < 120) become part of program code -- hard to add new constraints or change existing ones
- Failures may leave data in an inconsistent state with partial updates carried out

Databases 101

What is the biggest overhead in a database?



Speed and cost decreases while size increases from top to bottom

Parallel Selection

- Compute

  SELECT * FROM R

  SELECT * FROM R
  WHERE R.A < v1 AND R.A>v2

  SELECT * FROM R
  WHERE R.A =v

- What is the cost of these operations on a conventional database?
  - Cost = B(R)
- What is the cost of these operations on a parallel database with P processors?

Parallel Selection: Round Robin

tuple T_i to chunk (i mod P)

(1) SELECT * FROM R

(2) SELECT * FROM R
WHERE R.A = v

(3) SELECT * FROM R
WHERE R.A < v1 AND R.A>v2

- Best suited for sequential scan of entire relation on each query. Why?
- All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks
- Range queries are difficult to process --- tuples are scattered across all disks

Parallel Selection: Hash Partitioning

(1) SELECT * FROM R
(2) SELECT * FROM R
WHERE R.A = v

(3) SELECT * FROM R
WHERE R.A < v1 AND R.A>v2

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks
- Good for point queries on *partitioning attribute*
  - Can lookup single disk, leaving others available for answering other queries.
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Parallel Selection: Range Partitioning

(1) SELECT * FROM R

(2) SELECT * FROM R
WHERE R.A = v

(3) SELECT * FROM R
WHERE R.A < v1 AND R.A>v2

- Provides data clustering by partitioning attribute value
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries
- Caveat: badly chosen partition vector may assign too many tuples to some partitions and too few to others

Parallel Join

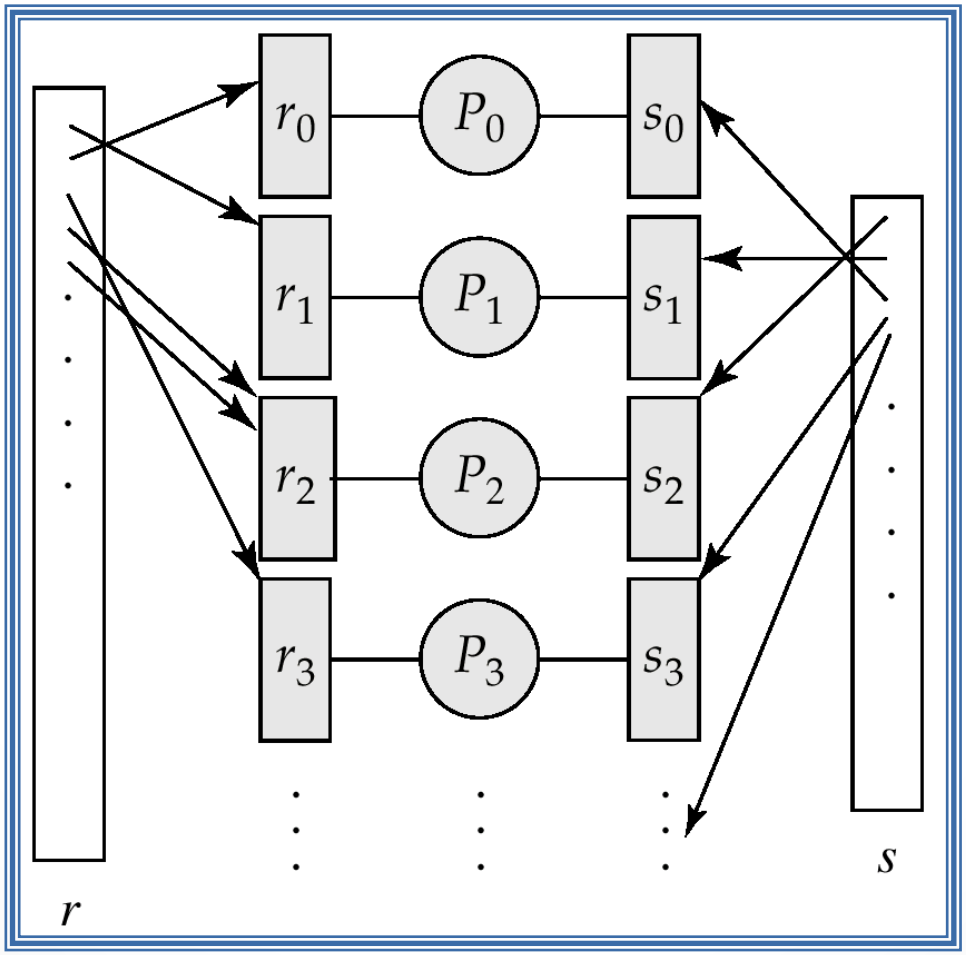
- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

How would you implement a join in MapReduce?

Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.
- *r* and *s* each are partitioned into *n* partitions, denoted r_0, r_1, ..., r_{n-1} and s_0, s_1, ..., s_{n-1}.
- Can use either *range partitioning* or *hash partitioning*.
- *r* and *s* must be partitioned on their join attributes *r*.A and *s*.B, *using the same range-partitioning vector or hash functio*n.
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$. Any of the standard join methods can be used.

Partitioned Join (Cont.)



Pipelined Parallelism

- Consider a join of four relations
  - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
- Set up a pipeline that computes the three joins in parallel
  - Let P1 be assigned the computation of
    temp1 = $r_1 \bowtie r_2$
  - And P2 be assigned the computation of
    temp2 = temp1 $\bowtie$ r_3
  - And P3 be assigned the computation of temp2 $\bowtie$ r_4
- Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
  - Provided a pipelineable join evaluation algorithm (e.g., indexed nested loops join) is used

Pipelined Parallelism

- Can we implement pipelined joins in MapReduce?

Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it *avoids writing intermediate results to disk*
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g., blocking operations such as aggregate and sort)
- Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

MapReduce: A Step Backwards

Dewitt and Stonebraker Views

- *We are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications*
- 1. A giant step backward in the programming paradigm for large-scale data intensive applications
- 2. A sub-optimal implementation, in that it uses brute force instead of indexing
- 3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
- 4. Missing most of the features that are routinely included in current DBMS
- 5. Incompatible with all of the tools DBMS users have come to depend on

http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

Dewitt and Stonebraker Views (cont.)

- The database community has learned the following three lessons over the past 40 years
  - Schemas are good.
  - Separation of the schema from the application is good.
  - High-level access languages are good
- MapReduce has learned none of these lessons
- MapReduce is a poor implementation
- MapReduce advocates have overlooked the issue of skew
  - Skew is a huge impediment to achieving successful scale-up in parallel query systems
  - When there is wide variance in the distribution of records with the same key lead some reduce instances to take much longer to run than others → execution time for the computation is the running time of the slowest reduce instance.

Dewitt and Stonebraker Views (cont.)

- I/O bottleneck: *N* map instances produces *M* output files
- If *N* is 1,000 and *M* is 500, the map phase produces 500,000 local files. When the reduce phase starts, each of the 500 reduce instances needs to read its 1,000 input files and must use a protocol like FTP to "pull" each of its input files from the nodes on which the map instances were run
- In contrast, Parallel Databases do not materialize their *split* files
- …

Case for Parallel Databases

Pavlo et al., SIGMOD 2009

MapReduce vs. Parallel Databases

- [Pavlo et al., SIGMOD 2009] compared the performance of Hadoop against Vertica and a Parallel DBMS
  - http://database.cs.brown.edu/projects/mapreduce-vs-dbms/
- Why use MapReduce when parallel databases are so efficient and effective?
- Point of view from the perspective of database researchers
- Compare the different approaches and perform an experimental evaluation

Architectural Elements: ParDB vs. MR

- Schema support:
  - Relational paradigm: rigid structure of rows and columns
  - Flexible structure, but need to write parsers and challenging to share results
- Indexing
  - B-trees to speed up access
  - No built-in indexes --- programmers must code indexes
- Programming model
  - Declarative, high-level language
  - Imperative, write programs
- Data distribution
  - Use knowledge of data distribution to automatically optimize queries
  - Programmer must optimize the access

Architectural Elements: ParDB vs. MR

- Execution strategy and fault tolerance:
  - Pipeline operators (push), failures dealt with at the transaction level
  - Write intermediate files (pull), provide fault tolerance

Architectural Elements

	Parallel DBMS	MapReduce
Schema Support	✓	Not out of the box
Indexing	✓	Not out of the box
Programming Model	Declarative (SQL)	Imperative (C/C++, Java, …) Extensions through Pig and Hive
Optimizations (Compression, Query Optimization)	✓	Not out of the box
Flexibility	Not out of the box	✓
Fault Tolerance	Coarse grained techniques	✓

[Pavlo et al., SIGMOD 2009, Stonebraker et al., CACM 2010, …]

Experimental Evaluation

- 5 tasks --- including task from original MapReduce paper
- Compared: Hadoop, DBMS-X, Vertica
- 100-node cluster
- Test speedup using clusters of size 1, 10, 25, 50, 100 nodes
  - Fix the size of data in each node to 535MB (match MapReduce paper)
  - Evenly divide data among the different nodes
- We will look at the Grep task --- see paper for details on the other tasks

Load Time

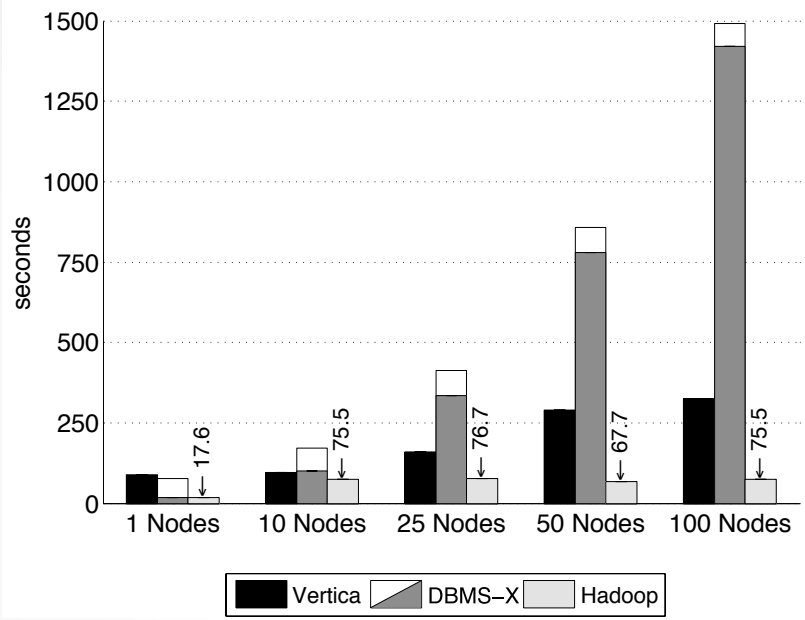


Figure 1: Load Times – Grep Task Data Set (535MB/node)

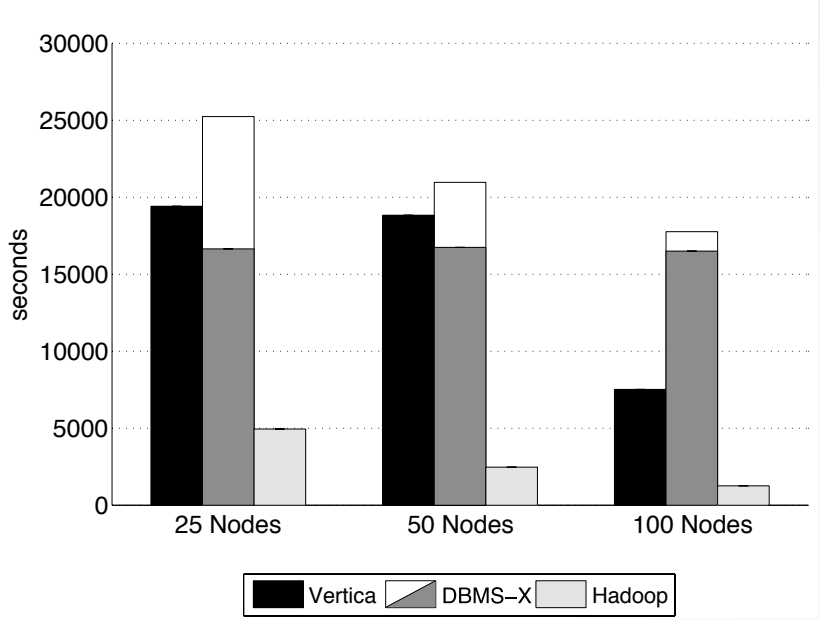


Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

- Hadoop outperforms both Vertica and DBMS-X

Grep Task

- Scan through a data set of 100-byte records looking for a three-character pattern. Each record consists of a unique key in the first 10 bytes, followed by a 90-byte random value.

SELECT * FROM Data WHERE field LIKE ‘%XYZ%’

Grep Task: Analysis

- Fig 4: Little data is processed in each node --- start-up costs for Hadoop dominate

"that takes 10–25 seconds before all Map tasks have been started and are running at full speed across the nodes in the cluster"

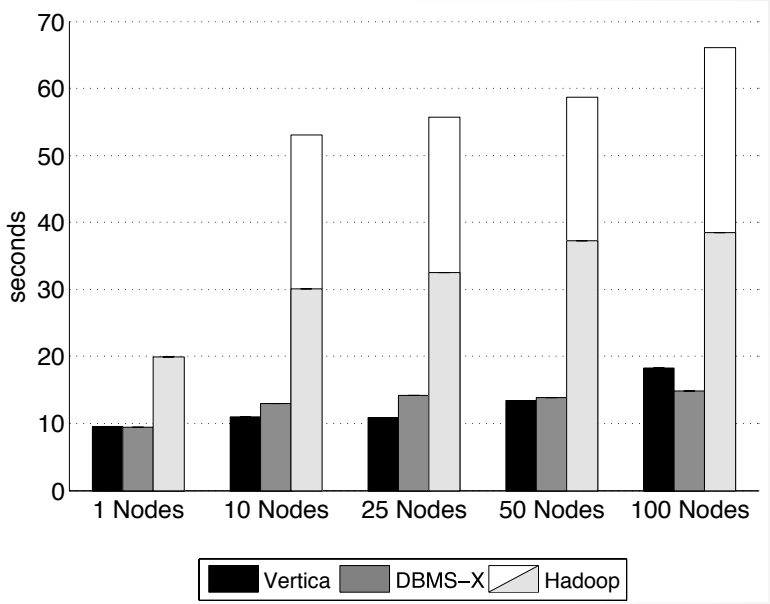


Figure 4: Grep Task Results – 535MB/node Data Set

Grep Task: Analysis

- Fig 5: Hadoop's start-up costs are ammortized --- more data processed in each node
- Vertica's superior performance is due to aggressive compression

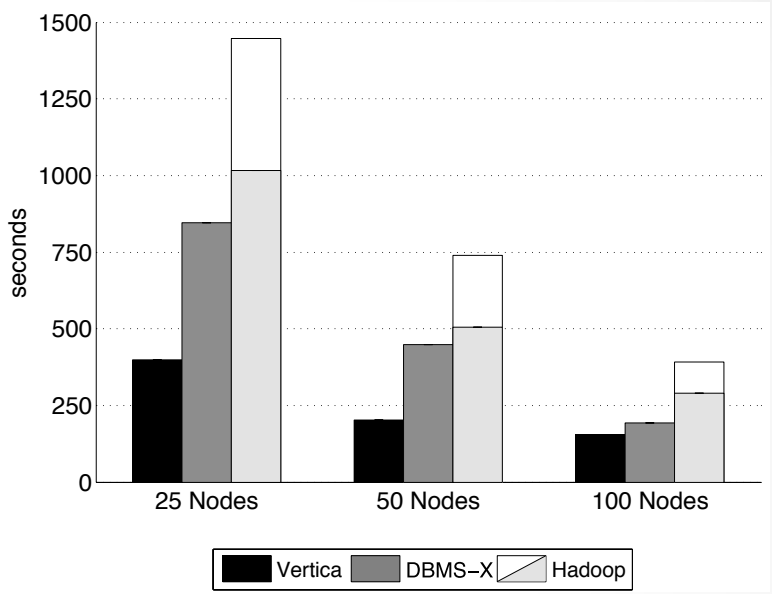


Figure 5: Grep Task Results – 1TB/cluster Data Set

Discussion

- Installation, configuration and use:
  - Hadoop is easy and free
  - DBMS-X is *very hard* --- lots of tuning required; and very expensive
- Task start-up is an issue with Hadoop
- Compression is helpful and supported by DBMS-X and Vertica
- Loading is much faster on Hadoop --- 20x faster than DBMS-X
  - If data will be processed a few times, it might not be worth it to use a parallel DBMS

Case for MapReduce

By Dean and Ghemawat, CACM 2010

MapReduce vs. Parallel Databases

- [Dean and Ghemawat, CACM 2010] criticize the comparison by Pavlo et al.
- Point of view from the creators of MapReduce
- Discuss misconceptions in Pavlo et al.
  - MapReduce cannot use indices
  - Inputs and outputs are always simple files in a file system
  - Require inefficient data formats
- MapReduce provides storage independence and fine-grained fault tolerance
- Supports complex transformations

Heterogeneous Systems

- Production environments use a plethora of storage systems: files, RDBMS, Bigtable, column stores
- MapReduce can be extended to support different storage backends --- it can be used to combine data from different sources
- Parallel databases require all data to be loaded
  - Would you use a ParDB to load Web pages retrieved by a crawler and build an inverted index?

Indices

- Techniques used by DBMSs can also be applied to MapReduce
- For example, HadoopDB gives Hadoop access to multiple single-node DBMS servers (e.g., PostgreSQL or MySQL) deployed across the cluster
  - It pushes as much as possible data processing into the database engine by issuing SQL queries (usually most of the Map/Combine phase logic is expressible in SQL)
- Indexing can also be obtained through appropriate partitioning of the data, e.g., *range partitioning*
  - Log files are partitioned based on date ranges

Complex Functions

- MapReduce was designed for *complex* tasks that manipulate diverse data:
  - Extract links from Web pages and aggregating them by target document
  - Generate inverted index files to support efficient search queries
  - Process all road segments in the world and rendering map images
- These data do not fit well in the relational paradigm
  - Remember: SQL is not Turing-complete!
- RDMS supports UDF, but these have limitations
  - Buggy in DBMS-X and missing in Vertica

Structured Data and Schemas

- Schemas are helpful to share data
- Google's MapReduce implementation supports the Protocol Buffer format
- A high-level language is used to describe the input and output types
  - Compiler-generated code hides the details of encoding/ decoding data
  - Use optimized binary representation --- compact and faster to encode/decode; huge performance gains – 80x for example in paper!

Protocol Buffer format

Quick Example

You write a `.proto` file like this:

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Then you compile it with `protoc`, the protocol buffer compiler, to produce code in C++, Java, or Python.

Then, if you are using C++, you use that code like this:

```
Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");

fstream out("person.pb", ios::out | ios::binary | ios::trunc);
person.SerializeToOstream(&out);
out.close();
```

Or like this:

```
Person person;
fstream in("person.pb", ios::in | ios::binary);
if (!person.ParseFromIstream(&in)) {
  cerr << "Failed to parse person.pb." << endl;
  exit(1);
}

cout << "ID: " << person.id() << endl;
cout << "name: " << person.name() << endl;
if (person.has_email()) {
  cout << "e-mail: " << person.email() << endl;
}
```

http://code.google.com/p/protobuf/

Fault Tolerance

- Pull model is necessary to provide fault tolerance
- It may lead to the creation of many small files
- Use implementation tricks to mitigate these costs
  - Keep this in mind when writing your MapReduce programs!

Conclusions

It doesn't make much sense to compare MapReduce and Parallel DBMS: they were designed for different purposes!

You can do anything in MapReduce

it may not be easy, but it is possible

MapReduce is *free*, Parallel DB are *expensive*

Growing ecosystem around MapReduce is making it more similar to PDBMSs

Making PDBMSs elastic

Transaction processing --- MR supports 1 job at a time

Conclusions (cont.)

- There is a lot of ongoing work on adding DB features to the *Cloud* environment
  - Spark: support streaming
  - Shark: large-scale data warehouse system for Spark
    - SQL API

      https://amplab.cs.berkeley.edu/software/
  - HadoopDB: hybrid of DBMS and MapReduce technologies that targets analytical workloads
  - Twister: enhanced runtime that supports iterative MapReduce computations efficiently