Advanced Database Management Systems

Massively-Parallel Data Processing

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Massive Parallelism and Distribution (1)
Tera- to Petabyte-Scale Data Volumes

- Recall that the post-1990s success of parallel/distributed architectures for DBMSs was attributed to three main factors, viz.:
  - the ascendancy of the relational model,
  - the high-quality of commodity hardware and network components and
  - the enabling power of modern software mechanisms.
- All of these advances have continued into the 2000s and the latter two in particular underpin the cutting edge of massively-parallel architectures that are likely to dominate the future of tera- to petabyte-scale data management and beyond, viz.:
  - **massively-parallel data appliances**
  - **massively-distributed cluster schemes**

Massive Parallelism and Distribution (2)
Data Appliances

- Massively-parallel data appliances can be seen as scale-up designs.
- They pack software and hardware in a box, deploying preconfigured and tuned DBMSs onto preconfigured and tuned processors and disks wired by a preconfigured and tuned interconnect.
- They adopt optimization schemes based on special storage strategies to minimize reliance on indexed and maximize the use of highly-parallel, high-performance scans and hash joins in query plans.
- While they major on OLAP-style queries, they are SQL engines.
- Consistently with the read-only nature of the workload, some data appliances remain shared-disk designs.

Massive Parallelism and Distribution (3)
Cluster-Based Schemes

- Massively-parallel cluster-based schemes replace scale-up (bigger boxes) with scale-out (more boxes) for their design strategy.
- They are the latest realization of extreme shared-nothing with the twist of custom-built software infrastructure for merging and splitting as well as scheduling and fault-tolerance.
- While they major in un- and semi-structured data processing, they can be targets for SQL-based query workloads.

Another Critique of Classical DBMSs (1)
From OLTP to OLAP

- While classical DBMSs do well on on-line transaction processing (OLTP), they are less good at on-line analytical processing (OLAP).
- OLTP typically consists of workloads that mix updates and queries whose profiles and demands can be studied and optimized for in advance.
- OLAP workloads are mostly read-only and dominated by ad-hoc (or transient) requirements.
- In modern organizations, the competitive edge that is provided by advanced data management is moving from OLTP to OLAP.

Another Critique of Classical DBMSs (2)
Not Cheap to Acquire, Costly to Maintain, Too Much Pain for Not Much Gain

- It has often been observed that the total cost of ownership (TCO) for DBMSs is increasing quite significantly.
- Even if the acquisition component of TCO were not high, the administration and operation costs are high and rising.
- The latter tend to be higher as workloads become shorter-lived, as they are in OLAP.
- This means that the price/performance ratio for classical DBMSs in OLAP settings is high.

Another Critique of Classical DBMSs (3)
Data Appliances

- In order to keep TCO down, administration and operation costs need to be kept down.
- The notion of an **appliance** connotes precisely this works-out-of-the-box, plug-and-run ideal.
- The notion first emerged in network management, then was picked up in keyword-based search applications.
- An appliance can be seen as special-purpose hardware running special-purpose software for a well-defined class of computing tasks.
- Both the hardware and the software are preconfigured and tuned (and this is where appliance vendors compete and aim to make their money).

Data Appliances (1)
Major Characteristics (1)

- **Shared-nothing massively-parallel processing (MPP)**: In order to deliver performance, data appliances make heavy use of all the advances in parallel DBMS we have studied.
- **Simpler QEPs on more raw computing power**: In order to both avoid high administration and operation costs and improve raw performance, data appliances strongly favour a simpler space of QEPs and physical schemas and pack enough high-quality hardware and fine-tuned software to make them go faster than complex QEPs.

Data Appliances (2)
Major Characteristics (2)

- QEP simplification is founded on two characteristics:
  - **Index-light**: avoid indices which are costly to design, store, maintain and make use of without the highest-quality DBAs being in constant interaction with application designers.
  - **Scan-heavy**: avoid random disk accesses and ensure instead that sequential scans come close to potential speed-up by physical designs that are tuned to high-quality, specially-configured, fine-tuned hardware.
- The major technical challenges centre around balanced, efficient data movements, i.e., the dynamic management of intermediate result sets and their timely, fast delivery to capable, fast-performing nodes.

Data Appliances (3)
Major Types/Current Players

Type 0 appliances use customized hardware. They are distinctive in using custom chips (or field-programmable gate arrays) and aggressively push computation to them in order to obtain better performance.

Type 1 appliances are custom assemblies of standard hardware that do not make use of chip-level tuning.

Type 2 appliances preconfigure and tune standard software and hardware into a coherent, tightly-coupled unit component.

Type 0 Netezza

quasi-**Type 1** Teradata, IBM

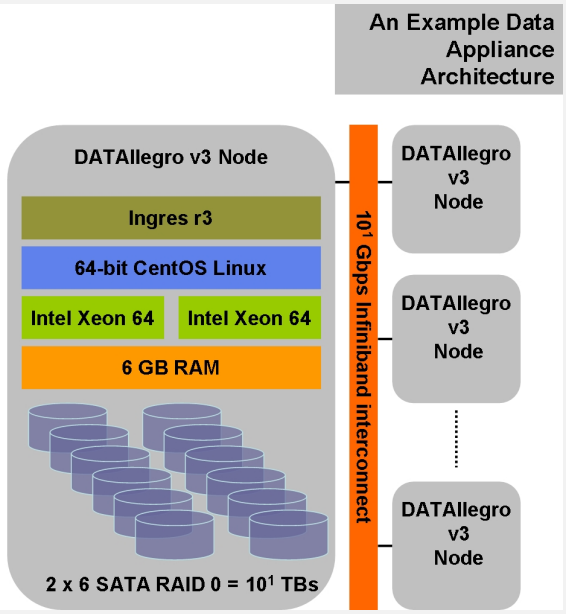
Type 2 DATAllegro

Data Appliances (4)
Raw Computing Power Needed

- Many, high-performance, high-capacity components, such as:
  - multiple top-end processors with excellent caching performance in the presence of massive amounts of RAM
  - multiple top-end storage units, with redundancy for reliability and for high-degree of parallelism
  - top-end interconnect to avoid slow down when in transit
- Balancing strategies to extract the best performance of such components

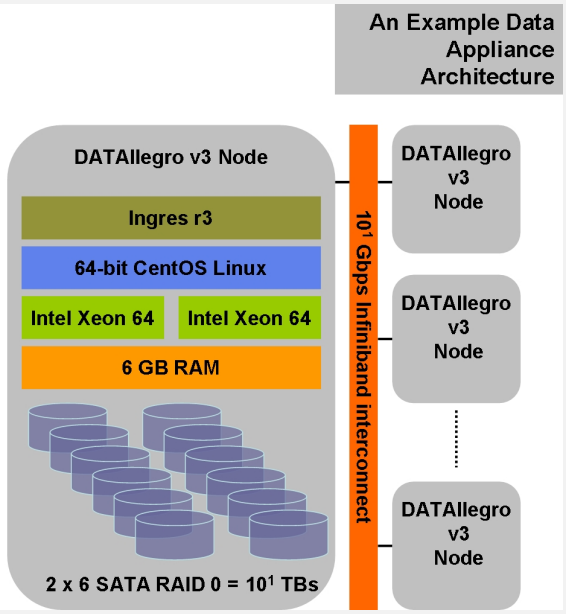
Data Appliances (5)
DATAllegro v3: A Type 2 Data Appliance (1)

- An appliance comprises multiple nodes, in a one-master, many-slaves relationship.
- The master node is responsible for query optimization, parallelization, concurrency, etc.
- Slaves node are responsible for storing data and executing the QEPs sent by the master node.



Data Appliances (6)
DATAllegro v3: A Type 2 Data Appliance (2)

- Each node runs an open source DBMS that is highly-, finely-tuned for the required query workloads (OLAP in the case of DATAllegro).
- Storage used a RAID (Redundant Array of Independent Drives) design intent on maximizing read performance.
- It uses top-end commodity components for each node and interconnects them with top-end network.
- If a node fails, another node picks up the role and load until it is slid out and a replacement is slid in.



Data Appliances (7)
Scan-Heavy Strategy

- As data is loaded, it is automatically hash-partitioned across the slave nodes.
- In addition, range- and hash-partitioning are used within a node to capitalize on the RAID storage.
- Tables can be replicated and/or partitioned.
- The query optimizer makes the decision between:
  - running the query within a node using replicas
  - running the query across nodes using partitioning
- The query optimizer is also fine-tuned to make it possible to use sequential scans and hash joins.
- Transfer units off storage are very large (e.g., 24-MB blocks).

Data Appliances (8)
Index-Light Strategy

- The index-light approach means that often one order-of-magnitude less storage space is needed, even without compression.
- The index-light approach implies not just less hardware but also less administration (to decide, design, maintain and make use of indices) and more consistent performance at a lower price/performance ratio.
- Rather than being slower at finding exactly the true positives, use raw power to read and process even the false positives faster.
- Indices can still be used: most data appliances are (or contain) a fully-fledged DBMS.

Summary
Massively-Parallel Data Processing

- A new breed of data management solutions has emerged recently that is a response to the new to lower TCO even in the presence of massive volumes of data and complex query workloads.
- The notion of a computing appliance first emerged in network management, then guided the development of integrated hardware/software products in the enterprise search market and is now being used to underpin cutting-edge solutions for OLAP querying.
- Data appliances used massively-parallel shared-nothing architectures for performance.
- They configure and tune the software and hardware to perform well with much reduced administration overheads.
- This is achieved by a combination of raw power and simplified, index-light, scan-heavy QEPs.

Advanced Database Management Systems
Massively-Distributed Data Processing (1)

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Data Appliances v. Cluster-Based Schemes (1)
TCO, Again

- Data appliances are perceived as having succeeded in reducing the administration and operation components of TCO, but their acquisition costs if in the order of tens of thousands per terabyte.
- Even though, by definition, appliances are designed, sold and deployed as an integrated hardware/software solution, there are two components to their acquisition cost:
  - the hardware, i.e., the top-end components that combine to produce the raw computing power without which the simplified QEPs would not deliver performance, and
  - the software, i.e., the partitioning and scheduling components that ensure that computing power is made the most use of and that constitute the competitive ground for vendors

Data Appliances v. Cluster-Based Schemes (2)
Major Types

Type 0 appliances use customized hardware. They are distinctive in using custom chips (or field-programmable gate arrays) and aggressively push computation to them in order to obtain better performance.

Type 1 appliances are custom assemblies of standard hardware that do not make use of chip-level tuning.

Type 2 appliances preconfigure and tune standard software and hardware into a coherent, tightly-coupled unit component.

Data Appliances v. Cluster-Based Schemes (3)
Clusters as "Type 3" Appliances

- Cluster-based approaches can be seen as an attempt to move onwards from Type 2 data appliances just as Type 2 appliances moved from Type 1 and Type 0 ones, i.e., in the direction of non-proprietary technologies.
- Thus, cluster-based schemes
  - can deliver performance even without specially configured and tuned assemblies of top-end hardware and network components
  - use open-source partitioning and scheduling software
- In this way, cluster-based schemes promise to lower the acquisition component of TCO and not just the administration and operation ones.
- While data appliances could be argued to be scale-up designs, cluster-based schemes are scale-out designs.

Cluster-Based Data Management

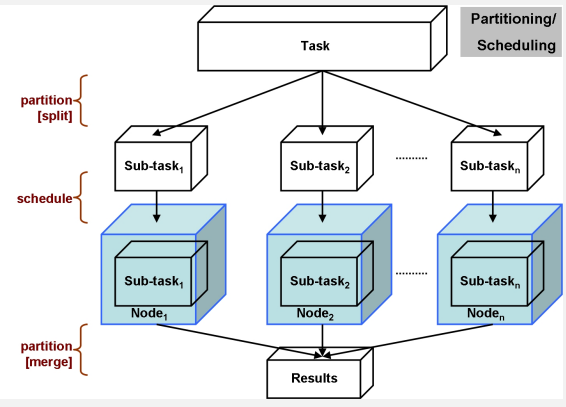
Advanced for Keyword-Based Querying, Incipient for Anything More Complex

- For partitioning and scheduling to be simple enough to carry out in a generic fashion, the parallel and distributed tasks must be even more stringently constrained than relational operators.
- The tasks involved in keyword-based search are an example.
- The tasks involved in frame-based image processing are another.
- In this respect, one of the major difficulties in query processing is data dependency (e.g., partitioning is sometimes attribute- and location-sensitive).

Cluster-Based Computing (1)

Partitioning and Scheduling (1)

- The grain of parallelization depends on the grain at which we can split a task into multiple independent units.
- The scheduling problem then is to assign sub-tasks to processing elements (i.e., threads, or nodes).
- Once sub-tasks are finished, the partitions are merged back into complete results.



Cluster-Based Computing (2)
Partitioning and Scheduling (2)

- The decision space is complex, and requires answers to the following questions (among others):
  - What if we cannot split the tasks into independent sub-tasks?
  - How do we assign sub-tasks to nodes?
  - What if we have more sub-tasks than nodes?
  - How do we merge sub-results into results?
  - How do we know all the nodes have finished?
- The common difficulty in all these is the need for nodes to communicate and access shared resources.
- It is crucial to achieve task independence by avoiding sharing state.

Cluster-Based Computing (3)
Partitioning and Scheduling (3)

Example (Expression-Level Independence)

- When independent branches of an algebraic expression commute and do not access (i.e., make no reference to) shared state, they can be executed in any order.
- In the following expression

  ```
  x := (a * b) + (y * z)
  ```

  since + commutes, a compiler (or a CPU scheduler) can execute the left- and the right-hand side in any order.

Example (Function-Level Independence)

- This is also true, but harder to automate, at the level of entire functions.
- In the following expression

  ```
  x := foo(a) + bar(b)
  ```

  if both `foo(_)` and `bar(_)` do not access shared state, they can be parallelized, with the assignment to `x` becoming the merge/synchronization point that blocks waiting for them to complete.

Cluster-Based Computing (4)
Detecting Dependencies Automatically is Hard

- In the last example, there is independence between `foo(_)` and `bar(_)` and dependence between each one of them and `x`.
- For a partitioner/scheduler infrastructure to detecting these properties is hard.
- The inability to do so prevents massive parallelization at the coarse grains that are needed to process massive volumes of data.

Cluster-Based Computing (5)
When It Can Be Done

- When task-level independence is ensured, in cluster-based computing (as well as in data appliances), a single-master, many-slaves approach can be employed:
  1. The master initially owns the data.
  2. The master spawns several workers to process the partitions it creates.
  3. The master waits for the workers to complete before either merging results.
- Daisy-chaining leads to workers that consume from and produce for other workers.
- Queues can be used to allow $n : m$ relationships between producing and consuming workers.
- In a distributed cluster, location transparency for both tasks and data is necessary.

Cluster-Based Computing (6)
But How Can It Be Done?

- One approach to circumventing this obstacle is to limit the expressiveness of the computation performed by individual tasks in such a way as to ensure task independence.
- One computational model that has acquired prominence recently (due to its widespread use in Google, Yahoo and other web-based companies) is referred to as **the map-reduce model**.

The Map-Reduce Computational Model (1)
Roots in Functional Programming (1)

- The map-reduce model takes its name from its roots in the functional programming paradigm (of which concrete examples are Haskell, Standard ML and Miranda, among others).
- In (pure) functional programs, functions are side-effect free.
- Recall our previous examples: in functional languages, two functions used in the same expression are guaranteed not to share any state.
- Instead of side-effecting shared data structures, functions in pure functional programs create new ones.
- They are, therefore, inherently parallelizable.

The Map-Reduce Computational Model (2)
Roots in Functional Programming (2)

- Consider a function `foo` that takes a list of numbers, computes a sum of its elements and adds to the latter the length of the input list.
- The following definition (in Haskell syntax) uses the `sum` and `length` primitives:

```
>> let foo x = sum x + length x
>> foo [1,2,3]
9
```

- Now, consider a function `bar` that takes a list of numbers and computes the square of applying `foo` to it.
- The definition (in Haskell syntax) could be:

```
>> let bar x = foo x * foo x
>> bar [1,2,3]
81
```


The Map-Reduce Computational Model (3)
Roots in Functional Programming (3)

- Since neither `foo` (or the primitives it relies on) nor `bar` interfere with another, if they appear as operands of a binary commutative function (such as +), they can be evaluated in any order:

```
>> foo [1,2,3] + bar [1,2,3] == bar [1,2,3] + foo [1,2,3]
True
```

- We can define a function append that takes an element and a list and appends the element to the end of the list in the following (convoluted way) without ever relying on shared state:
  1. Reverse the input list
  2. Stick the input element at the front of the results
  3. Reverse the latter result and return
- The definition in Haskell could be:

```
>> let append l ls = reverse (l : rev) where rev = reverse ls
>> append 1 [3,2]
[3,2,1]
```

The Map-Reduce Computational Model (4)
Roots in Functional Programming (4)

- Another important characteristic of functional programming languages is that functions can be passed as parameters to other functions.
- In particular, we can define **second-order functions**, i.e., functions that take functions as arguments.
- For example, we can define a function that takes a function and an argument, and applies the former to the latter twice.

```
>> let twice f x = f (f x)
>> twice (+ 1) 1
3
>> twice (* 3) 3
27
```

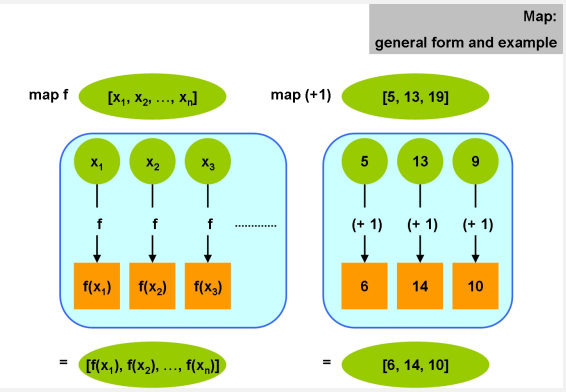
- In the examples, `(+ 1)` is a function that adds 1 to the argument it is applied to, and `(* 3)` is a function that triples to the argument it is applied to.

The Map-Reduce Computational Model (5)
Roots in Functional Programming (5)

- The map-reduce computational model has this name because it is centred around two higher-order functions that are well-known and well-studied in functional programming, viz., **map** and **fold**.
- Map takes as arguments a unary function and a list, applies the former to each element of the latter, and returns the resulting list.
- Fold takes as arguments a binary and associative function, a value (interpreted as the initialization of an accumulator) and a list.
- It traverses the list applying the input function to each element and the current value of the accumulator, updating the latter each time.
- It returns the final value in the accumulator once it reaches the end of the list.

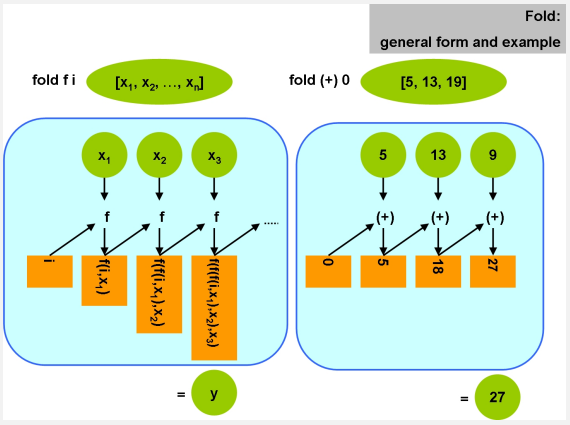
The Map-Reduce Computational Model (6)
Map and Fold(s): Examples (1)

- The general form taken by the application of a mapper function to a list is shown in the figure.
- To the right, a function `(+ 1)` that increments the elements of a list by one is applied.
- Note that the function could be Boolean-valued (i.e., a predicate), e.g., `(> 3)`.
- Does this suggest $\sigma_{x>3}(R)$ to you?



The Map-Reduce Computational Model (7)
Map and Fold(s): Examples (2)

- The general form taken by the application of a reducer function to a list is shown in the figure.
- To the right, a function `(+)` that adds the elements of a list (with the initializer set to zero) is applied.
- Does this suggest $\gamma_{SUM}(R)$ to you?
- The fold operation has two versions: `foldl` moves left-to-right in the list, `foldr` moves right-to-left.
- They are equivalent for commutative operations (e.g., addition), but not for non-commutative ones (e.g., subtraction).



The Map-Reduce Computational Model (8)
Map and Fold(s): Examples (3)

```
>> map (+ 2) [3, 4, 3, 8, 4, 1]
[5,6,5,10,6,3]
>> map (> 3) [3,2,4,6,8]
[False,False,True,True,True]
>> foldl (+) 0 [3, 4, 3, 8, 4, 1]
23
>> sum [3, 4, 3, 8, 4, 1] == foldl (+) 0 [3, 4, 3, 8, 4, 1]
True
```


The Map-Reduce Computational Model (9)
Map and Fold(s): Examples (4)

```
>> foldl (+) 0 (map (+ 2) [3, 4, 3, 8, 4, 1])
35
>> foldl (*) 1 (map (+ 2) [3, 4, 3, 8, 4, 1])
27000
```

The Map-Reduce Computational Model (10)
Map and Fold(s): Examples (5)

```
>> foldl (*) 1 [3, 4, 3, 8, 4, 1]
1152
>> sum [3, 4, 3, 8, 4, 1] + foldl (*) 1 [3, 4, 3, 8, 4, 1]
1175
>> foldl (*) 1 [3, 4, 3, 8, 4, 1] + sum [3, 4, 3, 8, 4, 1]
1175
>> (sum [3, 4, 3, 8, 4, 1] + foldl (*) 1 [3, 4, 3, 8, 4, 1])
    == (foldl (*) 1 [3, 4, 3, 8, 4, 1] + sum [3, 4, 3, 8, 4, 1])
True
>> (foldl (-) 0 [3, 4, 3, 8, 4, 1]) == ((((((0-3)-4)-3)-8)-4)-1)
True
>> (foldr (-) 0 [3, 4, 3, 8, 4, 1]) == (3-(4-(3-(8-(4-(1-0))))))
True
>>
```


The Map-Reduce Computational Model (11)
Implicit Parallelism in `map`

- In functional programming, applying the mapper function to an element does not have any impact on the result of applying the same mapper function to any other element
- If the order of application of the mapper is immaterial, it can be applied in parallel.
- The map-reduce computational model exploits this to the full.
- This is the way in which it limits the expressiveness of the computation performed by individual tasks so as to ensure task independence and hence be free execute tasks in massively-parallel mode.

Summary
Cluster-Based Schemes for Massively-Distributed Data Processing

- Cluster-based schemes hold the promise of going one step further in terms of commoditization than Type-2 data appliances.
- By constraining the computational model, it is possible to run massively-parallel computations without application developers having to design their solutions in parallel terms.
- One such computational model, founded on functional programming, centres around the well-known, well-studied **map** and **fold** higher-order functions.

Advanced Database Management Systems
Massively-Distributed Data Processing (2)

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Map-Reduce Engines (1)
Motivation

- The motivation is to process terabytes of data in parallel across thousands of processing elements without requiring the application developer to write any other code than mapper and reducer functions.
- The best-known map-reduce engine is Google's MapReduce/GFS.
- GFS stands for Google File System, which is the distributed storage-level infrastructure relied upon in Google's MapReduce.
- Yahoo has contributed an open-source version to the Apache Software Foundation, called Hadoop/DFS.

Map-Reduce Engines (2)
Functionalities

- A map-reduce engine automatically parallelizes and distributes the execution of the mapper and reducer functions that characterize an application.
- Hence, it is an infrastructure for automatic partitioning and scheduling.
- It is highly-fault-tolerant, i.e., it is extremely resilient when nodes (or tasks in nodes) fail.
- It provides monitoring tools and exports a simple (if restricted) abstraction to application developers.

Map-Reduce Engines (3)
The Programming/Execution Model (1)

- The programming model (i.e., the abstractions with which an application developer designs a solution to a problem) uses the functional programming approach we described above.
- Developers implement two functions with the following interfaces:

```
map ( in_key, in_value                    )
 -> (out_key, intermediate_value) list

reduce (out_key, intermediate_value list)
    -> out_value list
```

Map-Reduce Engines (4)
The Programming/Execution Model (2)

- The mapper function is fed data items (e.g., lines out of a file, rows out a database) in the form of (in_key, in_value) pairs.
- The role of the mapper function is to transform each one of the latter into an (out_key, intermediate_value) pair.
- Note that the map-reduce engine requires the mapper to:
  1. apply to one element only
  2. associate to the result an output key
- The results of the map phase are held in a barrier.
- By removing from the writer of the mapper function, any control over the grain of the task, a map-reduce engine can take decisions on task partitioning and parallel scheduling.

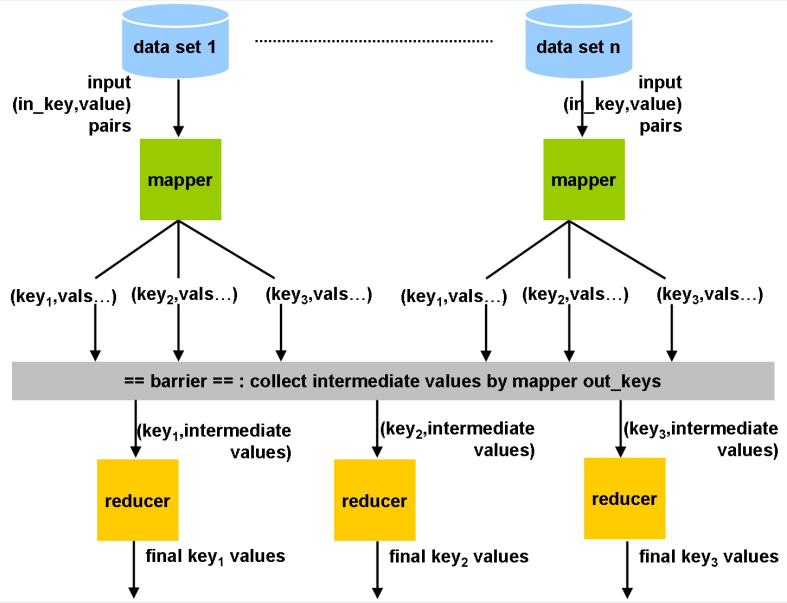
Map-Reduce Engines (5)
The Programming/Execution Model (3)

- The engine waits for the map phase to finish.
- When it is, all the intermediate values for a given output key are available in the barrier where they are combined to form a list per output key.
- The reducer function is fed such lists into one or more (typically one) final value for that same output key.
- By causing the writer of the mapper function to assign an output key, a map-reduce engine creates an opportunity to parallelize the reduction phase too, since each reduction task can operate on lists associated with different output keys.

Map-Reduce Engines (6)
The Programming/Execution Model (4)

- As far as implicit parallelization goes:
  - the mapper functions run in parallel, creating different intermediate values from different input data sets and emitting them with an output key, with which the engine can parallelize the reduce functions;
  - the reduced functions run in parallel, operating on different output key values.
- The only bottleneck is that the reduce phase cannot complete until the map phase is completed.

Map-Reduce Engines (7)
An Example Map-Reduce Computation



Map-Reduce Engines (8)

Example Mapper and Reducer from Google: Counting Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```


Map-Reduce Engines (9)

Locality Effects

- The master program divvies up tasks based on where data is located.
- For example, map tasks preferably run on the same node where the data file is, or at least in the same rack.
- Map tasks are given 64-MB blocks (which is the size of the chunks served by the underlying Google File System).

Map-Reduce Engines (10)
Fault Tolerance

- The master detects worker failures and
  1. Re-executes completed and in-progress map tasks.
  2. Re-executes in-progress reduce tasks.
- The master detects when an (in_key, in_value) pair causes a map task to crash and skips that when re-executing.

Map-Reduce Engines (11)
Some Optimizations

- Since the reduce task cannot complete until the map task is complete, a single slow component can rate-limit the whole process.
- Therefore, the master replicates slow-going map tasks in several components, keeps the first to finish and ignores the rest.
- So-called combiner tasks can be scheduled in the same machine as a mapper to perform a mini-reduce and save bandwidth before the real reduce phase starts.

Map-Reduce in Query Processing (1)
Early Days

- It is too early to say whether map-reduce will have an impact in query processing.
- There are, as yet, only preliminary accounts [Yang et al., 2007] as to how a map-reduce engine could act as a fully-functional query processing engine.
- Such accounts suggest that more than a mapper and a reducer functions may be needed.
- However, as hinted above, selection and group-by aggregation seem to translate fairly directly to map-reduce computations.
- So, for at least some OLAP workloads, a map-reduce engine may yet come to compete with data appliances.

Map-Reduce in Query Processing (2)
A Worked-Out Example (1)

- Assume that a relation with the following schema:

  ```
  Employees (emp_no:int, dept_name:str, salary:real)
  ```

  has been horizontally fragmented per branch name and that the latter has domain 'sales', 'manufacturing.
- Thus, the fragments are:

$$Employees_1 \leftarrow \sigma_{dept\_name='sales'}(Employees)$$

$$Employees_2 \leftarrow \sigma_{dept\_name='manufacturing'}(Employees)$$

  and, therefore, `Employees` can be reconstructed as:

$$Employees \leftarrow Employees_1 \cup Employees_2$$

Map-Reduce in Query Processing (3)
A Worked-Out Example (2)

- Consider now the following SQL query over the distributed database above:

```
SELECT    SUM(A.salary)
FROM      Employees A
WHERE     A.salary > 1000
GROUP BY  A.dept_name
```


Map-Reduce in Query Processing (4)
A Worked-Out Example (3)

```
map(String input_key, TupleSet input_value):
//          input_key: a fragment-defining attribute value
//        input_value: the tuples in that fragment
//         output_key: the GROUP BY attribute value
// intermediate_value: aggregation attribute value
//                     that satisfies the WHERE clause
for each tuple T in input_value:
  if  T.salary > 1000
       EmitIntermediate(T.dept_name, T.salary);

// the barrier contains all (group-name, group-member) pairs
```

Map-Reduce in Query Processing (5)
A Worked-Out Example (4)

```
// reduce gets (group-name, group-member) pairs from the barrier
// as if hashed buckets

reduce(String output_key, Iterator intermediate_values):
  //   output_key: the group-by attribute value
  // output_value: the partition contents defined by it
  int result = 0;
  for each v in intermediate_values:
    result += v;
Emit((output_key, result));
```


Map-Reduce in Query Processing (6)
A Worked-Out Example (5)

- The analysis of the above functions in terms of relational algebra is as follows.
- In this case, the mapper implements the following query:

$$B \leftarrow \pi_{dept\_name,salary}(\sigma_{salary>1000}(Employees_j))$$

- Because `Employees` is fragmented on `dept_name`, the parallelization criterion has already been made explicit, as it were.
- In relational terms, the barrier holds the result of running the above query in parallel over each fragment $j \in \{1, 2\}$.

Map-Reduce in Query Processing (7)
A Worked-Out Example (6)

- The reducer implements the following query:

$$V \leftarrow_{dept\_name} \gamma_{sum(salary)}(B)$$

- The result produced by each reducer instance running in parallel is a (single tuple) vertical fragment of the final result of the query, so to obtain the latter we use an iterated union:

$$\bigcup_{k \in dom(dept\_name)} V_k$$

Summary
Massively-Distributed Data Processing

- Massively-distributed processing over commodity clusters has proved itself in settings where the tasks that are parallelized have constrained expressiveness.
- Currently, it is an open question whether a map-reduce engine of the kind used by Google and Yahoo can be used for the kind of complex query workloads that data appliances are being used for.
- If so, massively-parallel cluster-based schemes could come to be seen as Type 3 appliances, with an appealing TCO proposition in all of acquisition, administration and operation.

Advanced Database Management Systems

Peer-to-Peer Infrastructure

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Internet-Scale Processing
The Other Dimension to Internet-Scale Data Volumes

- We have briefly looked into how the explosion in the scale of data volumes has sparked a renewed interest in massively-parallel and massively-distributed computational models.
- Such models have been deployed to process both enterprise-wide and Internet-scale data.
- One kind of architecture to make use of Internet-scale processing that may yet evolve into a mature DBMS technology is **peer-to-peer (P2P)**.

The Peer-to-Peer Approach (1)
Origins

- In Internet context, P2P architectures first gained attention in **file-sharing** applications such as Napster.
- A second appearance in the spotlight was started by **cycle-sharing** applications (typically based on screensavers) such as SETI@Home.
- Such early uses were P2P both in the sense of peer-to-peer and in the sense of point-to-point.
- This characteristic is not a problem for applications that are centred on end-users (as file sharing is) or on a single master (as cycle-sharing is).

The Peer-to-Peer Approach (2)
Focus

In the data management area, the P2P approach focusses on the following aspects:

- It is not a response to massive amounts of data, but rather an attempt to capitalize upon the availability of massive number of nodes in a network (thus, meganodes, not petabytes).
- Nodes are not equally capable, but they play identical roles (thus, heterogeneity but with symmetry).
- Minimal (or no) management of the distributed infrastructure.

The Peer-to-Peer Approach (3)
Challenges Relative to Other Parallel/Distributed Systems

- Partial failure is common, so the goal cannot be correctness and completeness but best effort.
- There are high-levels of churn, so dynamic (re)configuration and (re)sourcing is the norm.
- Fewer guarantees can be relied upon on transport, storage, etc..
- The optimization space is truly huge.
- Network bottlenecks and other resource constraints are beyond the reach of controls.
- No administrative organization to resort to.
- Trust issues are truly challenging.

The Peer-to-Peer Approach (4)
From Point-to-Point to Overlay Networks (1)

- An overlay network is a computer network which is built on top of another network.
- Nodes in the overlay network are connected by logical edges.
- Each single logical edge may map to a sequence of physical edges (i.e., a path) in the underlying network.
- For example, dial-up Internet is an overlay upon the telephone network.
- The early point-to-point approach to P2P applications evolved to the extent that such applications now use P2P networks that overlay networks: they run on top of the Internet.
- KaZaA and Gnutella are examples of this.

The Peer-to-Peer Approach (5)
From Point-to-Point to Overlay Networks (2)

- In overlay-based P2P application, peers do both **naming** and **routing** one-level up from the underlying network (typically, the Internet, in which case the latter is, conceptually, just a low-level transport layer, relative to the P2P application layer).
- The P2P application layer needs to:
  - keep track of the (IP) names and (IP) addresses of peers, which may be many and are subject to high-levels of churn
  - route messages among peers, which is multi-hop if (as is usually the case) peers do not keep track of every other peer.

The Peer-to-Peer Approach (6)
Unstructured v. Structured (1)

- An unstructured (or flat) P2P network is pure: all nodes are truly peers insofar as they all perform name and routing and have responsibilities whose scope is the set of all nodes.
- In a structured (or hierarchical) P2P network not all peers are equal: some peers are selected as ultra- (or super-) peers and take naming and routing responsibilities for a cluster of peers which it serves.
- The main motivation for structured P2P is that it avoids having to rely on flooding techniques over the entire network (only ultrapeers need getting involved).

The Peer-to-Peer Approach (7)
Unstructured v. Structured (2)

- Flood-based systems do well on:
  - Organic scaling
  - Simple global, and complex local, queries
  - Efficiently finding what is widely spread out
- Flood-based systems do not do so well on:
  - Efficiently finding what is not widely spread out
  - Complex global queries
- KaZaA and Gnutella are examples of structured P2P applications.

Data Independence (1)
The Original Insight

- Codd was the first to propose that decoupling the application-level interface from the concrete reality as to how data is organized and stored.
- The most beneficial consequence is that data organization and placement can change without forcing applications to change.
- The most basic level of this is location-independent logical names:
  - the name applications use is mapped to physical files
  - the application does not care where the latter reside: requests and responses are handled transparently
- From this basic level, the entire declarative-query processing stack can be built.

Data Independence (2)
The Recent Insight

- The extra level of indirection pays off if the underlying execution environment E does (or needs to) change more often than applications A do (or need to change).
- In terms of rates of change, the following must hold

$$\frac{d(E)}{dt} \gg \frac{d(A)}{dt}$$

- In classical data management, it holds because the right-hand side is unusually small (i.e., query-dominated applications change their queries slowly).
- In Internet-based networking, it holds because the the left-hand side is unusually large (i.e., the environment is highly dynamic).
- This motivates using indirection to achieve data independence over networks too.
- The network becomes the database.

Data Independence (3)
Enabling Techniques

- **Indexes**
  - Value-based look-ups have to compete with direct access
  - Indexes must adapt to shifting data distributions and must guarantee scalable performance
- **Query Optimization**
  - Support for declarative queries beyond lookup/search is necessary for added-value tasks
  - Techniques must adapt to shifting data distributions and changes in the execution environment

DBMS	P2P
B-trees	Distributed Hash Tables (DHTs)
Join ordering	Across-query shared flows
Access-method selection	Tree-staged aggregation
etc.	etc.

Distributed Hash Tables (1)
High-Level Idea: Indirection

Indirection in space by means of **content-based** logical IDs and of routing towards those IDs.

- To find the node N from a node M, Muses a hash function to obtain the location of N and delegates to peers between M and N to route query or data from M to N.
- This makes the network **content-addressable**.
- Because of indirection, churn at the physical level is contained and does not contaminate the logical level: nodes can join and leave without cascading of consequences.

Distributed Hash Tables (2)
High-Level Idea: Indirection

Indirection in time by means of **soft state**.

- It is imperative to decouple send and receive.
- Persistence is required for that.
- One approach often used in distributed systems is to require the requester to assign a time-to-live (TTL) to every persistent object.
- If TTL decrements to zero, the object is implicitly disposed of (i.e., garbage collected).
- If the requester requires an extension, then it must explicitly extend the TTL of the object.
- The requester can also dispose of an object explicitly.
- This makes the logical data space dynamic, and means that answers are reduced to best-effort only.

Distributed Hash Tables (3)
What are they?

- A hash table is a data structure that maps keys to values.
- A distributed hash table is a distributed data structure that maps a key to a networked node holding the key.
- Every DHT node supports one single operation: given a key, route messages towards the node that holds that key.
- DHT-enabled networks support two operations:

  insert(key, value) : at the issuer node F, the process is started by means of which the request is routed towards the node T where *key* is to be held, and, once there, the *value* is inserted in T as $(key, value)$ pair.

  lookup(key) : at the issuer node F, the process is started by means of which the request is routed towards the node T where *key* is held,and, once there, the *value* is retrieved from T using the *key*.
- Both operations are $O(logN)$ on the number N of nodes.

Distributed Hash Tables (4)
Design Goals

An **overlay network** with:

- Flexible mapping of keys to physical nodes
- Small network diameter
- Small degree (fanout)
- Local routing decisions
- Robustness to churn
- Routing flexibility
- Decent locality

A **storage framework** with :

- Maintenance via soft state
- No hard guarantees on persistence

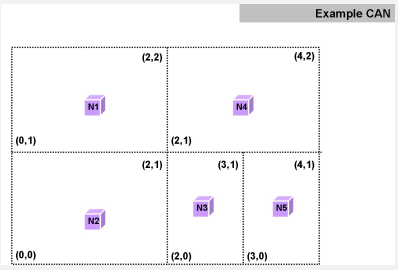
Distributed Hash Tables (5)
Peer v. Infrastructure

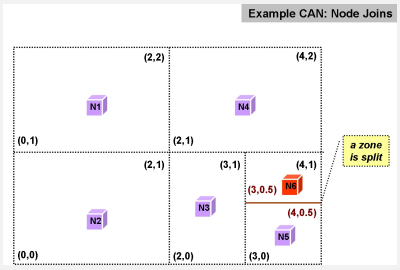
- Application users contribute nodes that make up the computational and data resources of a DHT-based network.
- The multi-application infrastructure emerges from the DHT-management code (as a library or as a service).

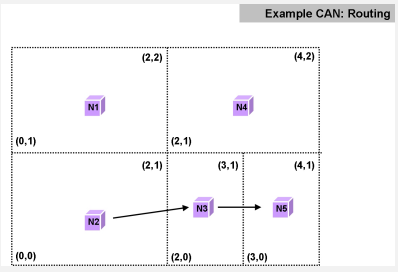
Distributed Hash Tables (6)
An Example DHT: CAN

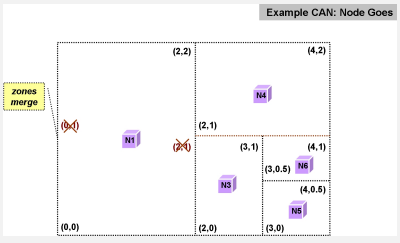
- There are many kinds of DHT (e.g., Chord, Koorde, Pastry, CAN, etc.).
- CAN (Content-Addressable Network) exploits the idea of multidimensionality to partition the logical space of IDs.
- Each node is assigned a zone and the boundaries of the zone determines the identity of the node.
- Thus, in two-dimensional space, a zone is a rectangle and a node is identified by the bottom-left and top-right corners of its zone.
- When a node joins the P2P network, a zone is split; when a node leaves two zones are merged.

Distributed Hash Tables (1)
An Example CAN in 2D









Distributed Hash Tables (2)
Some Observations

- Churn is contained within the neighbourhood of the joining/leaving node.
- Neighbours could be far apart in the underlying network.
- Generate routing tables (RTs) that result in few hops and low latency.
- (Algorithms exist that produce RTs with such good properties.)
- RT contains $O(d)$ neighbours.
- Routing is navigating in d-dimensional space.
- Number of hops is $O(dN^{\frac{1}{d}})$ for N nodes.

Summary
Peer-to-Peer Infrastructure

- The ubiquity of the Internet has meant that not only there are massive amounts of data to be processed, there are also massive numbers of nodes with which to do processing.
- P2P networks are overlay networks that attempt to capitalize on the opportunities arising from this fact.
- As P2P technologies evolved, DBMS researchers have proposed the idea that the notion of data independence is also valid in overlay networks.
- In particular, DHTs have been used to provide such independence, offer content-addressability and act as a storage layer.
- Seen in this light, DHT-based networks are databases, insofar as they can support complex querying, and not just keyword-based search.

Advanced Database Management Systems

Peer-to-Peer Query Processing

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Complex Queries (1)
Beyond Equality-Based Selection

- DHTs stop at equality-based selection.
- Support is also required for:
  - range predicates
  - aggregation and group-by aggregation
  - joins
- Evaluation strategies that are network-aware are also desirable.

Complex Queries (2)
The PIER Case

- PIER [Huebsch et al., 2005] is a P2P DHT-based query and storage engine that supports complex queries, as above.
- It relies fundamentally on DHTs and aggressively recasts the requirements above as functionalities to be supported over DHTs.
- There are alternatives to PIER, e.g., PeerDB [Ng et al., 2003] takes an agent-based view over nodes that are fully-fledged DBMSs.
- PIER (and PeerDB) refrains from projecting a global schema that integrates local ones, whereas PIAZZA [Halevy et al., 2004] does.

Complex Queries (3)
Supporting Range Queries

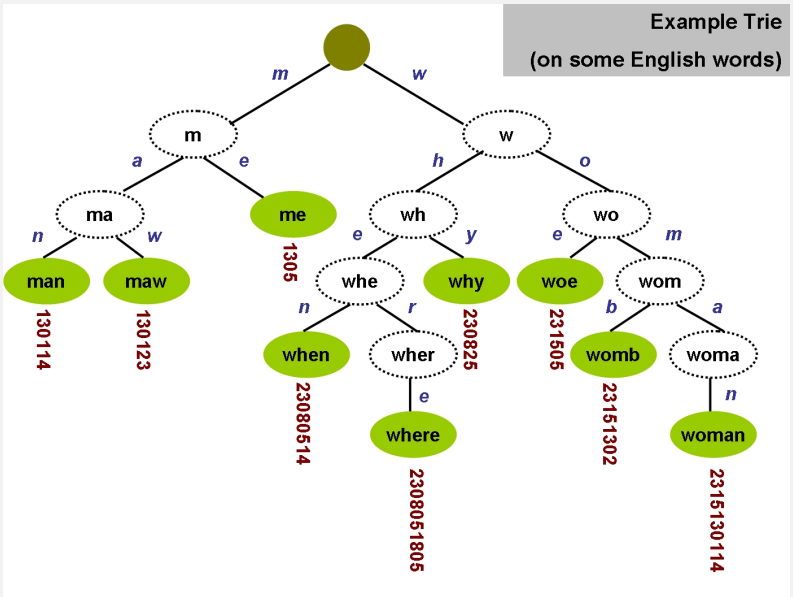
- The approach used in PIER to support range queries is based on prefix hash trees (PHTs).
- It has the advantage of working over any of the many types of DHTs.
- It gives rise to a directly-addressable distributed data structure.

Complex Queries (4)
PHTs from Tries (1)

- A **trie** (or **prefix tree**) is an ordered tree data structure that is used to store an associative array.
- Edges are labelled.
- No vertex in the tree stores the key associated with that vertex; instead, the path to a vertex is a candidate key.
- Not all vertices need to be associated with values, for those that are the candidate key is the key to the value.
- All the descendants of any one vertex have a common prefix associated with that vertex, and the root is associated with the empty string.

Complex Queries (5)
PHTs from Tries (2)

- In the figure, a vertex is labelled with its key.
- This key is the path to it.
- Nodes that have a dashed outline do not store values.
- Shaded vertices store the integer values shown below the vertex.



Complex Queries (6)
PHTs from Tries (3)

- A PHT is a binary bucket-based trie.
- In a PHT, every leaf vertex stores data.
- A data item K is stored in the unique vertex whose label is a prefix of K.
- A leaf vertex stores up to B items (i.e., each leaf vertex is a bucket with capacity B).
- To keep it balanced, a vertex is split if the B threshold is exceeded with the data items in the split vertex being partitioned among the children created for it.
- The novelty of PHTs is that they are a distributed data structure.
- The distribution is achieved by hashing the prefix labels of PHT vertices into the DHT identifier space.
- This hashing returns the P2P-network node where the bucket can be found.

Complex Queries (7)
PHTs from Tries (4)

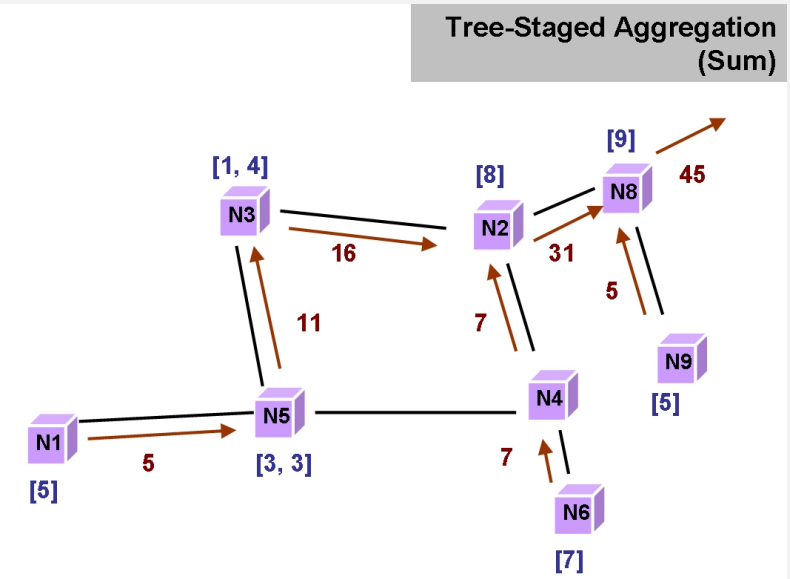
- This property of being directly accessible has desirable consequences.
- For example, given its prefix label, a PHT vertex can be located with a single DHT look-up.
- For PHT look-ups, binary search can be used rather than linear scans.
- Range queries involve a single PHT look-up followed by a sideways traversal of all leaf nodes whose prefixes overlap with the query.

Complex Queries (8)
Aggregation Queries (1)

- In this case too, DHTs offer an advantage.
- The fact that the overlay network is multi-hop means that even as intermediate nodes are doing the routing towards the destination, they can do some computation and thereby help reduced the bandwidth.
- Note, firstly, that conceptually the many data items will all travel towards the destination node.
- Along the route, they must converge on certain nodes, with the result that the overall form of the paths traversed will be that of a tree.
- At least at every confluence point (if not at each node), one can take the opportunity to do a partial aggregation and send that result forward.

Complex Queries (9)
Aggregation Queries (2)

- In the figure, the values held in each node are enclosed in square brackets.
- Arrows denote paths in the routing.
- Arrow labels show intermediate results for `SUM`



Complex Queries (10)
Group-By Aggregation

- In this case, each node sends tuples towards the hash ID of the grouping columns.
- An aggregation tree is still formed, but this time per group.
- Note that DHTs naturally induce a partitioning of the load, allowing for balanced demands over both the nodes and the network.

Complex Queries (11)
Hash Joins

- As we have seen, hash-based group-by is done very simply in DHT-based networks.
- Hash-based joins is, roughly, the same approach applied per input, i.e.:
  - Given $R \bowtie_{R.a=S.b} S$, then
  - Each node sends each R-tuple to $hash(R.a)$ and
  - Each node sends each S-tuple to $hash(S.b)$
- Again, trees are formed per hash destination.
- Although there may be lots of these trees, the DHT topology was pre-existing and no overhead is paid for that (although, of course, optimization opportunities always exist).
- Note also that no partial work can be done along the paths in the case of joins.
- Other join algorithms have also been proposed

Query Dissemination (1)
Broadcast Case

- One issue to address on P2P query evaluation is that of installing QEPs (or fragments thereof) in the nodes.
- If the query contains no indication of locality, then broadcasting the QEP (fragments) to all nodes in the path from the sources to the destination is needed.
- This is the opposite of forming an aggregation tree.
- One approach to the formation of this dissemination tree is to use flooding, i.e., each node passes on to its neighbours each query when it has received it.
- This is, however, wasteful (as a node may receive the same query multiple times).

Query Dissemination (2)
Unicast Case

- For a query of the form `SELECT <attr-list> FROM R WHERE <eq-pred-on-R-attr>`, if the attribute in the predicate has already been hashed into the DHT, then the QEP can be routed by the DHT just as data is.
- In this case, query dissemination follows an access path through an index, which is the DHT in its indexing role.
- If more equality predicates are used, sub-parts can be obtained and each can be disseminated using the same technique.

Summary
Peer-to-Peer Query Processing

- Several P2P systems have offered support for complex queries.
- DHT-based systems (like PIER) have also demonstrated that it is possible to do so with elegance and economy of means.
- Nonetheless, there is a need for a great deal of further work before such approaches prove robust and reliable enough for widespread deployment.
- In particular, and as expected of any distributed DBMS, P2P DBMSs will have to have answers to the problem of semantic integration, and awareness of this has spawned relevant work already.

Acknowledgements (1)

The material presented mixes original material by the author as well as material adapted from

- [Monash, 2007a, Monash, 2007b]
- [Hellerstein, 2004]

The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

Acknowledgements (2)

***

Portions of the material presented are modifications based on work created and shared by Google and used according to terms described in the Creative Commons 2.5 Attribution License:

http://creativecommons.org/licenses/by/2.5/

The original content was authored by Aaron Kimball, Sierra Michels-Slettvet, Christophe Bisciglia, Hannah Tang, Albert Wong, Alex Moshchuk et al. and can be accessed from

http://code.google.com/edu/content/parallel.html

The current author is grateful to Google and the authors of the original content for sharing their work. The current author assumes responsibility for any errors or mistakes introduced in the modifications made.

***

References

Halevy, A. Y., Ives, Z. G., Madhavan, J., Mork, P., Suciu, D., and Tatarinov, I. (2004).
The piazza peer data management system.
IEEE Trans. Knowl. Data Eng., 16(7):787–798.

Hellerstein, J. M. (2004).
Architectures and algorithms for internet-scale (p2p) data management.
VLDB 2004 Tutorial.
http://db.cs.berkeley.edu/jmh/talks/vldb04-p2ptut-final.pps.

Huebsch, R., Chun, B. N., Hellerstein, J. M., Loo, B. T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., and Yumerefendi, A. R. (2005).
The architecture of pier: an internet-scale query processor.
In CIDR, pages 28–43.

Monash, C. A. (2007a).
Design choices in mpp data warehousing.
Technical report, Monash Research White Paper.

Monash, C. A. (2007b).
Index-light mpp data warehousing.
Technical report, Monash Research White Paper.

Ng, W. S., Ooi, B. C., Tan, K.-L., and Zhou, A. (2003).
Peerdb: A p2p-based system for distributed data sharing.
In Dayal, U., Ramamritham, K., and Vijayaraman, T. M., editors, ICDE, pages 633–644. IEEE Computer Society.

Yang, H.-C., Dasdan, A., Hsiao, R.-L., and Jr., D. S. P. (2007).
Map-reduce-merge: simplified relational data processing on large clusters.
In Chan, C. Y., Ooi, B. C., and Zhou, A., editors, SIGMOD Conference, pages 1029–1040. ACM.