
Parallelizing Queries, Evaluating Queries

2 May 2010
Lecture 7

Topics for Today

- Parallel Query Evaluation
 - Concepts
 - Parallelizing Queries
 - Examples
- Introduction to Indexes
- Introduction to Query Evaluation
- Source: RG 22.1-22.2, 12

Introduction to Parallelization

- Serial Execution
 - Begin
 - Run
 - End
- Data Partitioning
- Skew
- Pipelining

What do we gain by it?

Algorithm a on data n

- Complete serial execution: $S_a(n)$
 - So working on *many* n_1, n_2, n_3, n_4 : We need to do $S_a(n_1) + S_a(n_2) + S_a(n_3) + S_a(n_4) + \dots$
- What if we could run a on n by breaking it into m pieces and running it on m computers?
 - Ideally: $S_a(n)/m$ runtime
- That's almost *never* the case, there is some setup and collecting time (may be relative to the size of n):
 - Setup time: $B_a(n)$
 - Completion time: $C_a(n)$
 - Our real run time: $B_a(n) + S_a(n)/m + C_a(n)$
- So we must check whether:

$$B_a(n) + S_a(n)/m + C_a(n) < S_a(n)$$

But that's not all... (1/2)

$$B_a(n) + S_a(n)/m + C_a(n) < S_a(n)$$

How do we divide up the job into m pieces?

- Do we ship the data to the m computers and then have them ship back the results?
 - Distribution time: $D_a(n)$
 - Recollection time: $R_a(n)$
 - So we must now check: $B_a(n) + D_a(n) + S_a(n)/m + R_a(n) + C_a(n) < S_a(n)$
-

- What if we don't do the division of work perfectly?
 - Say we made n_1, n_2, \dots, n_m pieces
 - Let $n_{max} = MAX(n_1, n_2, \dots, n_m)$
 - Then we really are checking:
$$B_a(n) + D_a(n) + S_a(n_{max}) + R_a(n) + C_a(n) < S_a(n)$$
 - That means we are benefitting only as much as we can reduce the size of the biggest piece of n
 - **Skew** refers to how fair the division is: $(n_{max} - n/m)$ (**less = better**)

But that's not all... (2/2)

$$B_a(n) + D_a(n) + S_a(n_{max}) + R_a(n) + C_a(n) < S_a(n)$$

- This still isn't realistic – most algorithms contain a portion which is not parallelizable
 - It may be a part which depends serially on its previous output
 - It may be some processing which must be done together
- Call the parallelizable segment: $P_a(n)$
- Call the rest $NP_a(n)$

So we really must check:

$$B_a(n) + D_a(n) + NP_a(n) + P_a(n_{max}) + R_a(n) + C_a(n) < S_a(n)$$

Pipelining

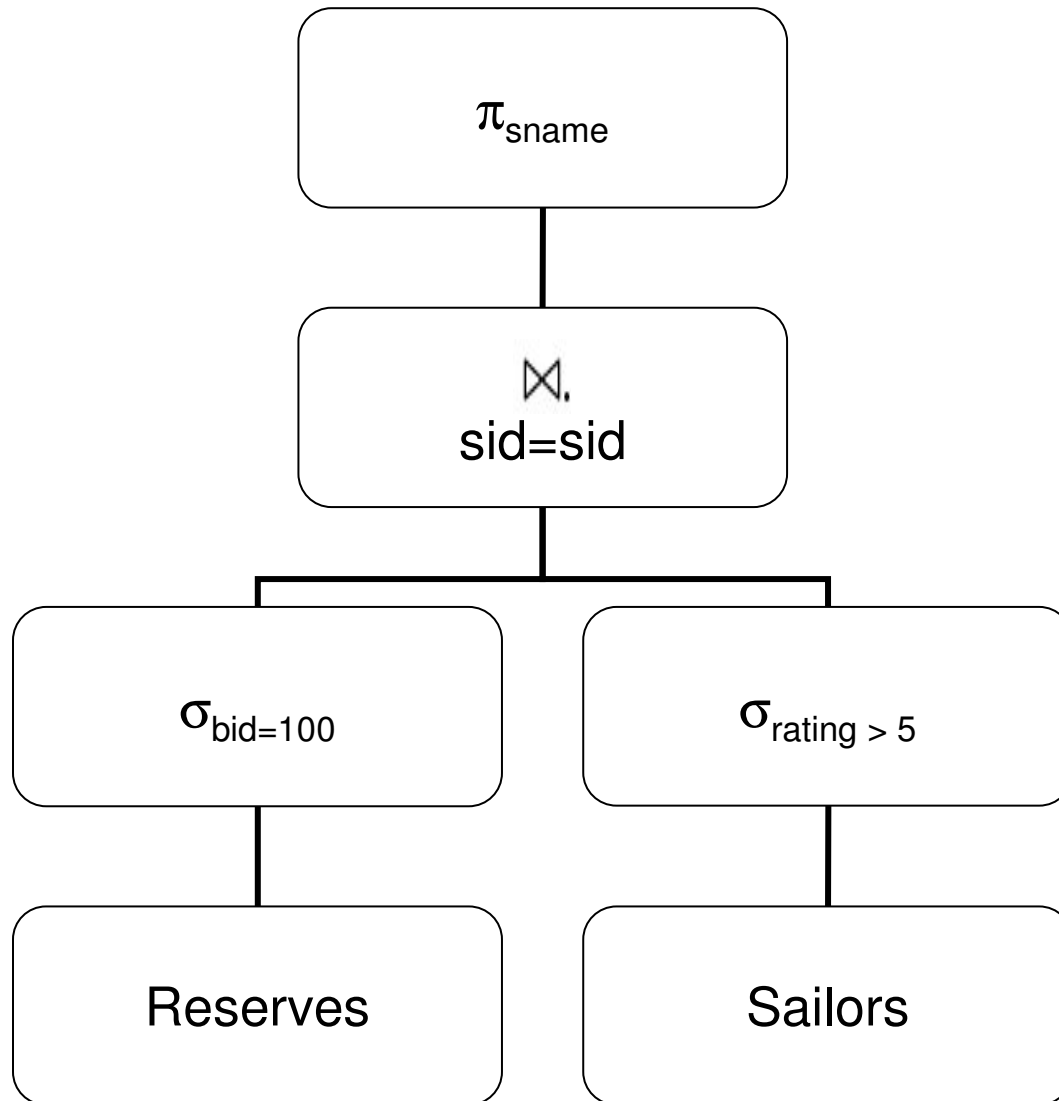
Another alternative: Pipelined evaluation

- Assumes that operations are **not atomic** and can be broken into pieces which are executed one at a time
- We break a into k parts: a_1, a_2, \dots, a_k
 - Have k workers do the job – worker j just does part a_j
 - Worker j does his job and then passes it along to worker $j+1$

Evaluating a Query

- How do we evaluate a query?
 - We didn't talk about this last semester
- A DBMS prepares an *Evaluation Plan*
 - Often it's based on an extended version of relational algebra
- A relational algebra expression can be rewritten as a tree
 - Each sibling can be evaluated in parallel
 - Parents use the output of their children

Evaluation Plan



```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
      AND R.bid = 100
      AND S.rating > 5
```

Evaluation Parallelism

- Parents consume the output of children
- Pipelined Parallelism
 - Feed as it comes
- Blocking
 - Must wait for the full results
 - Aggregation (SUM, AVG)
 - Sorting (but we can parallelize this a bit better)

Example Schema

Emp (*eid*:integer, *ename*:string, *age*:integer, *salary*:real)

Works (*eid*:integer, *did*:integer, *pct_time*:integer)

Dept (*did*:integer, *dname*:string, *budget*:real,
managerid:integer)

Example 1

```
SELECT ename, age
FROM Emp E1, Dept D1, Works W1
WHERE E1.eid = W1.eid
AND D1.did = W1.did
AND D1.dname = ``Hardware"
```

INTERSECT

```
(SELECT ename, age
FROM Emp E1, Dept D1, Works W1
WHERE E1.eid = W1.eid
AND D1.did = W1.did
AND D1.dname = ``Software")
```

Example 2

```
SELECT eid FROM Emp E1
WHERE 100 <
  (SELECT SUM(pct_time)
   FROM Works W2
   WHERE W2.eid = E1.eid);
```

Examples 3

```
SELECT MIN(AGE) FROM Emp E1 WHERE NOT EXISTS  
(SELECT did FROM Works W2
```

MINUS

```
(SELECT W3.did  
FROM Works W3  
WHERE E1.eid = W3.eid)  
)
```

Examples 4

```
SELECT AVG(avg_sal) average, num
FROM (
  SELECT AVG(salary) as avg_sal, COUNT(*) as num
  FROM Works W, Emp E
  WHERE E.eid = W.eid
  AND E.age < 18
  GROUP BY W.did
  HAVING COUNT(*) >= 2) byDept
GROUP BY num
```

Examples 5

```
SELECT SUM(budget) FROM Dept D1
WHERE NOT EXISTS (SELECT *
                  FROM Works W2, Emp E2
                  WHERE W2.did = D1.did
                  AND E2.eid = W2.eid
                  AND E2.age >= 18)
```

Data-Partitioned Parallel Evaluation

- How about parallelizing each *operator*?
 - Divide the work up among many processors
- First we need some background on how queries are evaluated at all in the database...

So Far

- Parallel Query Evaluation
 - Concepts
 - Parallelizing Queries
 - Examples
- Introduction to Indexes
- Introduction to Query Evaluation

Organizing data

- Data in a DBMS are stored in *files*
 - Each file has *pages* of data
- Many ways to organize the file (affects performance)
 - How is the file organized? (sorted? randomly?)
 - How are the pages arranged? (serially? randomly?)
 - How are the data arranged on the pages?

Example: Relation R(*name*:string, *age*:int, *salary*:real).

- If we organize the file by *age*, we can easily answer “Who is 20 years old?”
- To answer “Who makes 10,000 Shekels per month” we need to scan the whole relation.

Data and Files

- Most DBMS' don't have space in memory for all of its data
 - *Pages* are moved in/out of memory like Virtual Memory Paging Systems
 - The cost of reading/writing pages from disk a the biggest bottleneck a DBMS faces
 - Each record in a file has a unique *record id or rid*
 - The *rid* tells us which page the record is in
-
- Most common file organization is as a *File of Records*
 - File can be *created, destroyed, inserted into, deleted from, scanned* (going through each record one at a time)
 - An unordered File of Records is called a *Heap File*
 - Can return all records or just one with a specific *rid*

Improvement: Indexes

- How do we find out which employees are 20 years old?
 - In a heap file → scan the whole relation!
 - Better idea: **Index** which keeps track of where particular values can be found in the relation
 - The values that it indexes are the *search key*
 - Key could be one field (*age*) or many fields (*age, salary*)
-
- Let a data entry (row) in the index be written k^* if it refers to a row with search key value k
 - Three ways to organize an index:
 - 1) A data entry k^* is an **actual data record** (with the search key value k)
 - 2) A data entry is a (k, rid) pair where rid is the record id of a data record with the search key value k
 - 3) A data entry is a $(k, rid-list)$ pair where the *rid-list* is a list of record ids of data records with search key value k

Index Types

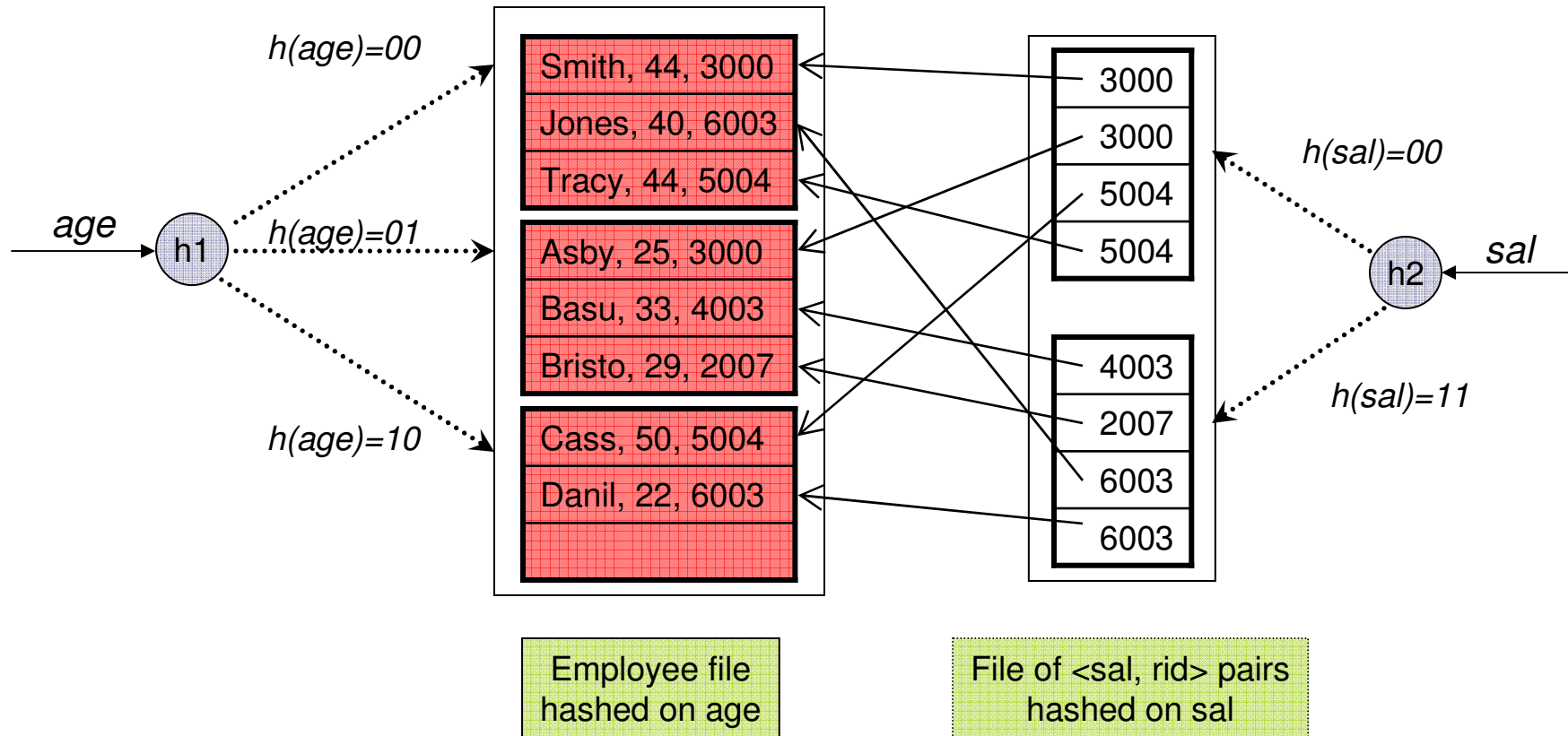
- Alternative (1) above is a *clustered index file*
 - Means the relation is **sorted** based on the index
 - Accessing records can be **much faster** this way (if selecting by the search key)
-

Indexes are commonly organized by either

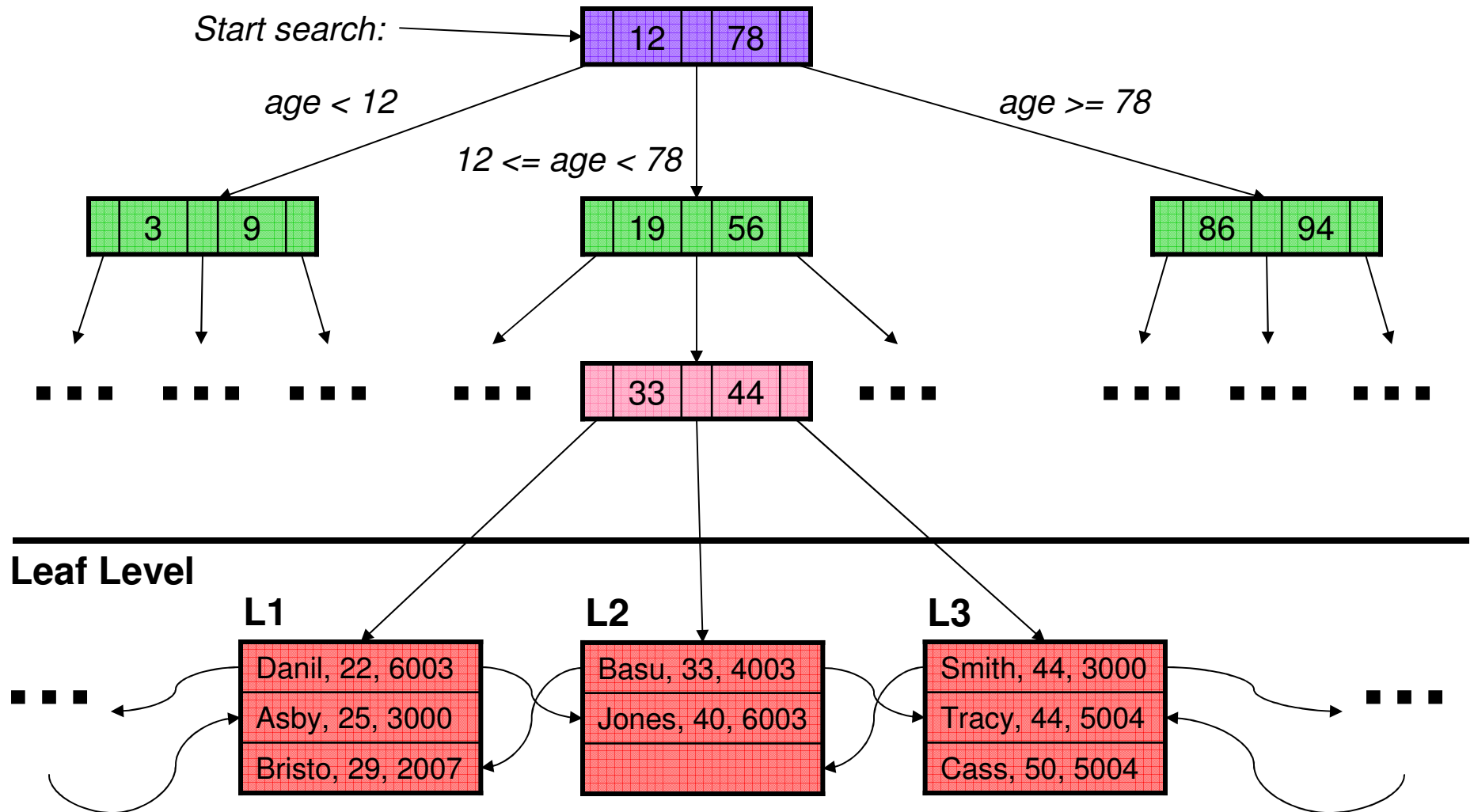
- **Hashing**
 - Hash the values for the search key to determine where the entry goes in the index
- **Tree based indexing**
 - Create a sorted search tree based on the values of the search key
 - Think of a binary search tree

These are *virtual structures* – in practice they are flat files with pointers to different *offsets/pointers* within the file

Index-Organized File Hashed on *age* with auxiliary index on *sal*



Tree-Structured Index



So Far

- Parallel Query Evaluation
 - Concepts
 - Parallelizing Queries
 - Examples
- Introduction to Indexes
- Introduction to Query Evaluation

Overview of Query Evaluation

- We saw SQL and Relational Algebra last semester
 - The two are (intentionally) closely related
 - Algebra is the key to understanding how queries are evaluated

For our examples, consider the relations

Sailors(*sid* :integer, *string*:rating, *rating*:integer, *age*:real)

Reserves(*sid* :integer, *bid*:int, *day*:datetime, *rname*:string)

Relation	Bytes/Tuple	Tuples/Page	# Pages	# Tuples
Reserves	40	100	1,000	100,000
Sailors	50	80	500	40,000

The System Catalog

- The data in the database → Relations, Indexes
- Database also stores **Metadata** → data about the relations, indexes
 - Also stored in tables
 - Can use SQL to browse it!
- Metadata tables are called **catalog tables**
 - Or **data dictionary, system catalog, catalog**

The Attribute_Cat Relation

<i>attr_name</i>	<i>rel_name</i>	<i>type</i>	<i>position</i>
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	datetime	3
rname	Reserves	string	4

The Catalog Stores...

System information

- Buffer pool size
- Page size

Per Table information

- Table name
- File name where it's stored
- Type of the file
- Attribute name and type for each field
- Index name of each index on the table
- Integrity constraints (keys, foreign keys)

Per Index information

- Index name and structure
- Search key attributes

Per View information

- View name
- View Definition

Other Table/Index Information (updated periodically)

- Cardinality of tables
($NTuples(R)$)
- Cardinality of indexes
($INPages(I)$)
- Size in pages of tables
($NPages(R)$)
- Size in pages of indexes
($INPages(I)$)
 - For trees, the number of leaf nodes, range, height

Conclusion

- Parallel Query Evaluation
 - Concepts
 - Parallelizing Queries
 - Examples
- Introduction to Query Evaluation