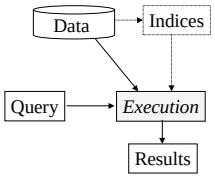
Query Execution in Databases: An Introduction

Zack Ives
CSE 544
Spring 2000

Role of Query Execution

- A runtime interpreter
  … or, the *systems* part of DBMS
- Inputs:
  - Query execution plan from optimizer
  - Data from source relations
  - Indices
- Outputs:
  - Query result data
  - Updated data distribution statistics (sometimes)



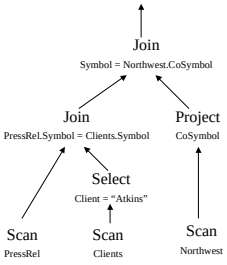
Outline

- Overview
- Basic principles
- Primitive relational operators
- Aggregation and other advanced operators
- Querying XML
- Trends in Execution
- Wrap-up: execution issues

Query Plans

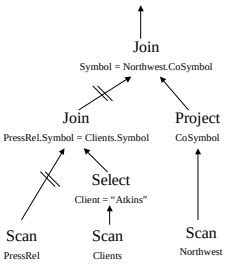
- Data-flow graph of relational algebra operators
- *Typically:* determined by optimizer
- *Trends*: adaptivity for distributed data

```
SELECT *
FROM PressRel p, Clients C
WHERE p.Symbol = c.Symbol
 AND c.Client = 'Atkins'
 AND c.Symbol IN
(SELECT CoSymbol FROM Northwest)
```



Execution Strategy Issues

- Granularity & parallelism:
  - Pipelining vs. blocking
  - Threads
  - Materialization
- Control flow:
  - Iterator/top-down
  - Data-driven/bottom-up
  - Threads?

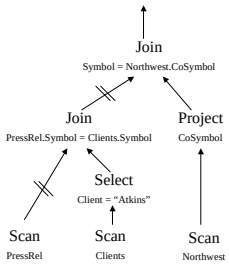


Data-Driven Execution

- Schedule via leaves
  (generally parallel or distributed system)
- Leaves feed data "up" tree; may need to buffer

Π Good for slow sources or parallel/distributed

O In typical system, can be inefficient

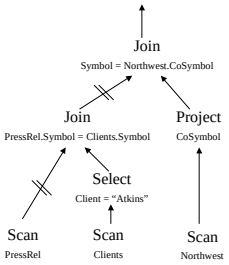


The Iterator Model

- Execution begins at root
  - *open, next, close*
  - Propagate calls to children
    - May call multiple child *next*s

Π Efficient scheduling & resource usage

If slow sources, children communicate from separate threads



7

Execution Has a Cost

- Different execution plans produce same results at different cost – optimizer estimates these
  - It must search for low-cost query execution plan
  - Statistics:
    - Cardinalities
    - Histograms (estimate *selectivities*)
- I/O vs. computation costs
- Pipelining vs. blocking operators
- Time-to-first-tuple vs. completion time

8

Basic Principles

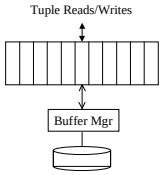
- Many DB operations require reading tuples, tuple vs. previous tuples, or tuples vs. tuples in another table
- Techniques generally used:
  - *Iteration*: for/while loop comparing with all tuples on disk
  - *Buffered I/O*: buffer manager with page replacement
  - *Index*: if comparison of attribute that's indexed, look up matches in index & return those
  - *Sort*: iteration against presorted data (*interesting orders*)
  - *Hash*: build hash table of the tuple list, *probe* the hash table

☛ *Must be able to support larger-than-memory data*

9

Reducing I/O Costs with Buffering

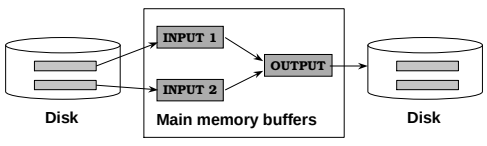
- Read a page/block at a time
- Should look familiar to OS people!
  - Use a page replacement strategy:
    - LRU (not as good as you might think)
    - MRU (good for one-time sequential scans)
    - Clock, etc.
  - *Note that we have more knowledge than OS to predict paging behavior*
    - DBMIN (min # pages, local policy)
- Double-buffering, striping common



10

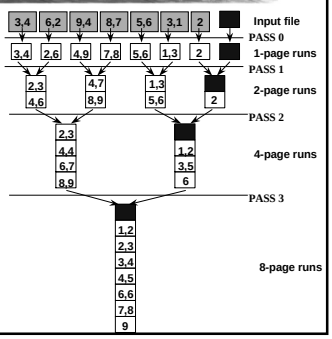
Two-Way External Sorting

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

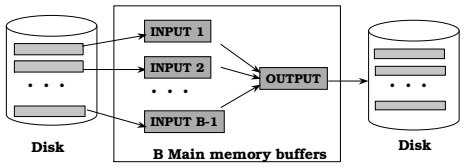
- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  $= \lceil \log_2 N \rceil + 1$
- Total cost is:
  $2N\left(\lceil \log_2 N \rceil + 1\right)$
- *Idea:* ***Divide and conquer:*** sort subfiles and merge



General External Merge Sort

☞ ***How can we utilize more than 3 buffer pages?***

- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
  - Pass 2, …, etc.: merge $B-1$ runs.



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- With 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Hashing

- A familiar idea:
  - Requires "good" hash function (may depend on data)
  - Distribute across buckets
  - Often multiple items with same key
- Types of hash tables:
  - Static
  - Extendible (requires directory to buckets; can split)
  - Linear (two levels, rotate through + split; bad with skew)

15

Basic Operators

- Select
- Project
- Join
  - Various implementations
  - Handling of larger-than-memory sources
- Semi-join

16

Basic Operators: *Select*

- If unsorted & no index, check against predicate:

```
Read tuple
While tuple doesn't meet predicate
  Read tuple
Return tuple
```

- Sorted data: can stop after particular value encountered
- Indexed data: apply predicate to index, if possible
- If predicate is:
  - conjunction: may use indexes and/or scanning loop above (may need to sort/hash to compute intersection)
  - disjunction: may use union of index results, or scanning loop

17

Basic Operators: *Project*

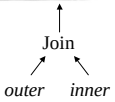
- Simple scanning method often used if no index:

```
Read tuple
While more tuples
  Output specified attributes
  Read tuple
```

- Duplicate removal may be necessary
  - Partition output into separate files by bucket, do duplicate removal on those
  - May need to use recursion
  - If have many duplicates, sorting may be better
- Can sometimes do index-only scan, if projected attributes are all indexed

18

Basic Operators: *Join* — Nested-Loops



- Requires two nested loops:

```
For each tuple in outer relation
  For each tuple in inner, compare
  If match on join attribute, output
```

- Block nested loops join: read & match page at a time
- What if join attributes are indexed?

  Index nested-loops join

- Results have order of outer relation

Π Very simple to implement

O Inefficient if size of inner relation > memory (keep swapping pages); requires sequential search for match

(Sort-)Merge Join

- Requires data sorted by join attributes
  - Use an external sort (as previously described), unless data is *already* ordered

```
Merge and join the files, reading sequentially a
  block at a time
```

  - Maintain two file pointers; advance pointer that's pointing at guaranteed non-matches
  - Preserves sorted order of "outer" relation

Π Allows joins based on inequalities (non-equijoins)

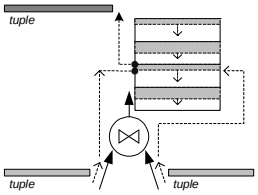
Π Very efficient for presorted data

O Not pipelined unless data is presorted

Hash-Based Joins

- Allows partial pipelining of operations with equality comparisons (e.g. equijoin, union)
- Sort-based operations block, but allow range and inequality comparisons
- Hash joins usually done with static number of hash buckets
  - Generally have fairly long chains at each bucket
  - Require a mechanism for handling large datasets

Hash Join




```
Read entire inner
  relation into hash
  table (join attributes
  as key)
For each tuple from
  outer, look up in hash
  table & join
```

Π Very efficient, very good for databases

O Not fully pipelined

O Supports equijoins only

O Delay-sensitive

Running out of Memory

- Two possible strategies:
  - Overflow prevention (prevent from happening)
  - Overflow resolution (handle overflow when it occurs)
- GRACE hash overflow resolution: split into groups of buckets, run recursively:

```
Write each bucket to separate file
Finish reading inner, swapping tuples to
  appropriate files
Read outer, swapping tuples to overflow files
  matching those from inner
Recursively GRACE hash join matching outer &
  inner overflow files
```

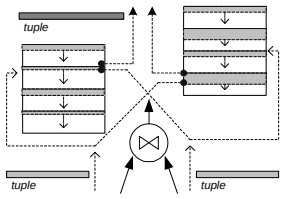

Hybrid Hash Join Overflow

- A "lazy" version of the GRACE hash:

```
When memory overflows, swap a subset of the
  tables
Continue reading inner relation and building
  table (sending tuples to buckets on disk as
  necessary)
Read outer, joining with buckets in memory or
  swapping to disk as appropriate
Join the corresponding overflow files, using
  recursion
```

Pipelined Hash Join

(a.k.a. Double-Pipelined Join, XJoin, Hash Ripple Join)



- Two hash tables
- As a tuple comes in, add to the appropriate side & join with opposite table

Π Fully pipelined, data-driven

O Needs more memory

Overflow Resolution in the DPJoin

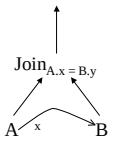
- Based on the ideas of hybrid hash overflow
- Requires a bunch of ugly bookkeeping!

  Need to *mark* tuples depending on state of opposite bucket - this lets us know whether they need to be joined later
- Three proposed strategies:
  - Tukwila's "left flush" – "get rid" of one of the hash tables
  - Tukwila's "symmetric" – get rid of one of the hash buckets (both tables)
  - XJoin's method – get rid of biggest bucket; during delays, start joining what was overflowed

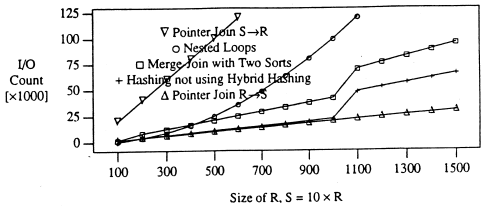
The Semi-Join/Dependent Join

- Take attributes from left and feed to the right source as input/filter
- Important in data integration
- Simple method:

  for each tuple from left
  send to right source
  get data back, join
- More complex:
  - Hash "cache" of attributes & mappings
  - Don't send attribute already seen
  - Bloom joins (use bit-vectors to reduce traffic)



Join Comparison



Aggregation + Duplicate Removal

- Duplicate removal equivalent to agg function that returns first of duplicate tuples
- Min, Max, Avg, Sum, Count over GROUP BY
  - Iterative approach: while key attribute(s) same:
    - Read tuples from child
    - Update value based on field(s) of interest
  - Some systems can do this via indices
  - Merge approach
  - Hash approach
- Same techniques usable for difference, union

What about XML?

- XML query languages like XML-QL choose graph nodes to operate on via *regular path expressions* of edges to follow:

```
WHERE <db><lab>
        <name>$n</>
        <_*.city>$c</>
      </> ELEMENT_AS $l </> IN "myfile.xml"
```

is equivalent to path expressions:

```
$l = (root).db.lab
$n = $l.name
$c = $l.(anything)*.city
```

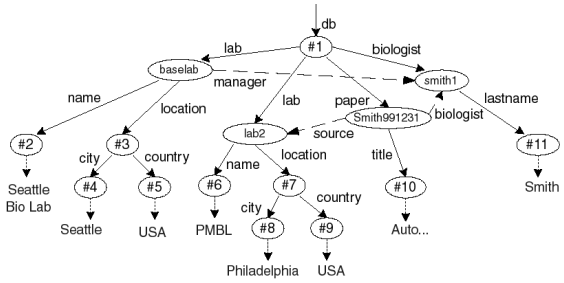
- We want to find tuples of (**`$l`**, **`$n`**, **`$c`**) values
- Later we'll do relational-like operations on these tuples (e.g. join, select)

Example XML Document

```
<db>
  <lab ID="baselab" manager="smith1">
   <name>Seattle Bio Lab</name>
   <location>
     <city>Seattle</city>
     <country>USA</country>
   </location>
  </lab>
  <lab ID="lab2">
   <name>PMBL</name>
   <city>Philadelphia</city>
   <country>USA</country>
  </lab>
  <paper ID="Smith991231" source="baselab" biologist="smith1">
   <title>Autocatalysis of Spectral...</title>
  </paper>
  <biologist ID="smith1">
    <lastname>Smith</lastname>
  </biologist>
 </db>
```

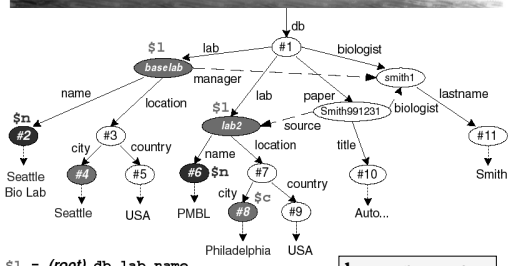
31

XML Data Graph



32

Binding Graph Nodes to Variables




```
$1 = (root).db.lab.name
$n = $1.name
$c = $1.(anything)*.city
```

$1	$n	$c
baselab	#2	#4
lab2	#6	#8

33

XML Operators

- New operators:
  - XML construction:
    - Create element (add tags around data)
    - Add attribute(s) to element (similar to join)
    - Nest element under other element (similar to join)
  - Path expression evaluation
    - X-scan

34

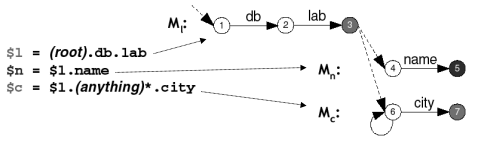
X-Scan: "Scan" for Streaming XML

- We often re-read XML from net on every query
  Data integration, data exchange, reading from Web
- Previous systems:
  - Store XML on disk, then index & query
  - Cannot amortize storage costs
- X-scan works on *streaming* XML data
  - Read & parse
  - Track nodes by ID
  - Index XML graph structure
  - **Evaluate path expressions to select nodes**

35

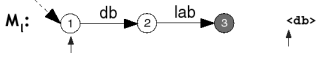
Computing Regular Path Expressions

Create *finite state machines* for path expressions



M_n and M_c are "dependent" on M_l

Start with initial (l) machine active:

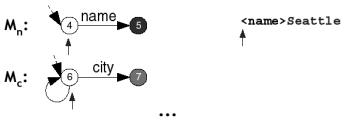


36

More State Machines

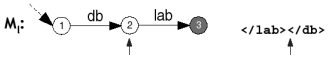
Read through XML to **lab** contents:

We now have a value for l; activate dependents:



...

Reset to previous state on each close-tag:



37

X-Scan works on *Graphs*

- The state machines work on trees – what about IDREFs?
  - Need to save the document so we can revisit nodes
  - Keep track of every ID
  - Build an "index" of the XML document's structure; add *real* edges for every subelement and IDREF
  - When IDREF encountered, see if ID is known
    - If so, dereference and follow it
    - Otherwise, parse and index until we get to it, then process the newly indexed data

38

Recent Work in Execution

- XML query processors for data integration
  - Tukwila, Niagara (Wisconsin), Lore, MIX
- "Adaptive" query processing – smarter execution
  - Handling of exceptions (Tukwila)
  - Rescheduling of operations while delays occur (XJoin, query scrambling, Bouganim's multi-fragment execution)
  - Prioritization of tuples (WHIRL)
  - Rate-directed tuple flow (Eddies)
  - Partial results (Niagara)
  - "Continuous" queries (CQ, NiagraCQ)

39

Where's Execution Headed?

- *Adaptive scheduling* of operations – not purely iterator or data-driven
- *Robust* – as in distributed systems, exploit replicas, handle failures
- Able to show and update *partial/tentative results* – operators not "fully" blocking any more
- More *interactive and responsive* – many non-batch-oriented applications
- More *complex data models* – handle XML efficiently

40

Leading into Our Next Topic: Execution Issues for the Optimizer

- Goal: minimize I/O costs!
- "Interesting orders"
- Existing indices
- How much memory do I have and need?
  Selectivity estimates
- Inner relation vs. outer relation
- Am I doing an equijoin or some other join?
- Is pipelining important?
- Good estimates of access costs?

41