Implementing Joins and Other Operations

Dan Fasulo

May 1, 1998

Review

Ways of storing data:

- Unsorted
- Sorted

Ways of indexing data:

- Sorted array
- Hash table
- B+ Tree
- clustered vs. unclustered

Defining Performance

- Principally interested in counting disk page I/O ops

- Sailors relation:
  - `(sid : int, name : string, boat : int, ...)`
  - $N = 500$ pages
  - $p_S = 80$ records per page

- Reservations relation:
  - `(rid : int, sid : int, restaurant : int, ...)`
  - $M = 1000$ pages
  - $p_R = 100$ records per page

Basic Joins

```
SELECT *
FROM Reservations R, Sailors S
WHERE R.sid = S.sid
```

- Defined as all pairs of tuples from R and S where `R.sid = S.sid`

- Can naively be thought of as cross product followed by selection

- Naive Nested Loop implementation

```
for each r in R
  for each s in S
    if (r.sid = s.sid) then output (r, s)
```

- $M + p_R * M * N$ (about 5 million!) I/O's

Page and Block Joins

- *Page-at-a-time*: instead of scanning S for each record r, scan it for each page of records in R

- Reduces cost to $M + M * N$

- If R fits in memory, we can do it in $M + N$ time!

- Even if not, suppose that we can put B pages of R in memory at a time.
  - Load B blocks of R for each scan of S: $M + \lceil \frac{R}{B} \rceil * N$ time
  - Alternatively, load $B/2$ blocks of each for slightly more I/O, but less seek time

- Optimize for equality by building hash table...

Sort-Merge Join

- Suppose we have an external sort function
- We can sort the relations, then run a mergesort-like merge process to select tuples
- Assuming that the number of matches is relatively low, takes time $M + N$ plus the sort time
- If number of matches is high, can take time $M * N$!
- Can be optimized to do sort and selection at the same time; gives expected time $3(M + N)$.

An Example

- Suppose we have 35 buffers
  - 4000 I/O's to sort Reservations
  - 2000 I/O's to sort Sailors
  - Additional scan to merge: 1500 I/O's
  - Block nested-loops: 15,000 I/O's

- Suppose we have 100 buffers
  - Sort-merge stays at 7,500 I/O's
  - Block nested-loops: 2,500 I/O's

Grace Hash Join

- Suppose that we have a hash function h which distributes both R and S evenly, and so that r and s go into the same partition if the selection criteria holds

- Idea: partition relations into k partitions; recursively join each partition
  - For each partition, read smaller of partition for S and partition for R into memory
  - Build hash table for in-memory partition

Hash Join Continued

- If no partition overflows, takes time $3(M + N)$
  - $2(M + N)$ to read relations and write partitions
  - Dealing with each partition in memory means we can read each page once, so $M + N$ time

- For good performance, we want the average partition size $\min(R, S)/k$, to be smaller than available memory

- If we have *much* more free memory, we can handle one or more partitions *while* we do the partitioning, thus avoiding writing them to disk

Hash vs. Sort-merge Join

- Assume that number of matches is "small"

- If we have $B > \sqrt{\text{min}(M, N)}$ buffer pages, then hash join takes time $3(M + N)$.

- If we have $B > \sqrt{\text{max}(M, N)}$ buffer pages, then sort-merge join takes time $3(M + N)$.

- Hash join is sensitive to the hash function

- Hash join is typically better if one relation is much smaller than the other

- Merge join is much cheaper if sorted relations are already available

- Merge-join outputs selected key in sorted order

Selection and Projection

- Selection: conceptually, based on search of data structure holding Relation

- Projection: The difficult bit is duplicate elimination; this is almost like doing a join on a relation with itself
  - Merge-based projection: sort projected elements, then sweep to remove duplicates
  - Hash based: Partition into k partitions with appropriate hash function; recursively remove duplicates from partitions