Lecture 11:
Database-related research at ITU

Rasmus Pagh

Background literature (not curriculum): [PP06 sec. 1-3], [PPR05 sec. 1-2], [FPP06 sec. 1-2], [PPT04 sec. 1-3], [BHPPRT06, sec. 1,3,4.0,4.1], [JP06, sec. 1.0, 1.1, 2.0, 2.1]

Today

- ITU research in databases:
  - Scalable computation of acyclic joins.
  - Binary joins - when can it be done faster?
  - Hashing on external memory:
    - Can it be improved?
    - Without a hash function?

- A brief overview of some main points of the course.

Problem session

- Recall the way in which the join of many relations is handled by traditional query optimizers.
- When does join computation go really bad?
- Try to think of the worst possible performance (I/O and space usage).

Acyclic joins

- **Intuition**. Consider the graph with:
  - Relations as vertices.
  - An edge between two vertices if there is an equality condition between the relations.
- This is called the *join graph*.
- If the join graph is *acyclic*, there is an efficient procedure for eliminating all ”dangling” tuples that will not contribute to the final result. (Blackboard.)

Complexity of acyclic join

- Suppose we want to join k relations of n blocks in total, and that the final result has size z blocks.
- What can we say about the I/O cost in general, for the classical query optimization approach?
- **Bad case:** Star schema. Computation may require $\Omega(zk)$ I/Os. (Blackboard.)

Counting-based algorithm [PP06]

- Discard the idea of computing the join as a sequence of binary joins.
- Instead, consider the whole join at once.
- **Case study (blackboard):**
  1. Star schema with foreign key constraints.
  2. General star join case:
     - Observation 1: We can efficiently compute how many occurrences there will be of each tuple.
     - Observation 2: We may efficiently assign ”output tuple numbers” to the occurrences of a tuple.
     - Result can be computed by sorting according to output tuple number.

Binary join computation

- You have seen two 2-pass join algorithms:
  - Sorting based. Space usage $\sqrt{B(R)+B(S)}$
  - Hashing based. Space usage $\sqrt{\min(B(R),B(S))}$
- Internal memory space usage may influence the ability to pipeline operations, evict pages from the internal memory buffer, etc.
- Can we improve the space, and/or efficiency?
  - In some cases, yes!

Method 1: Filtering [PPR05]

- **Idea (old)**: Eliminate all (or most) ”dangling” tuples that will not be part of the join result.
- Let S be the set of values in the join attribute of the smallest relation.
- Let h be a hash function that computes a short ”signature” of attribute values.
- In some cases, the set $S' = \{h(x) \mid x \in S\}$ can be stored internally. (Use 8 bits/tuple, say.)
- Then most dangling tuples (y,...) of the larger relation can be removed in a single scan since h(y) is not in S’.

Method 2: String sorting [FPP06]

- The time for sort join grows linearly with the size of the elements (strings) of the join attributes.
- **Idea:** Use recursion on the *length* of strings to quickly reduce the length of the strings considered.
- To sort
  we first recursively sort

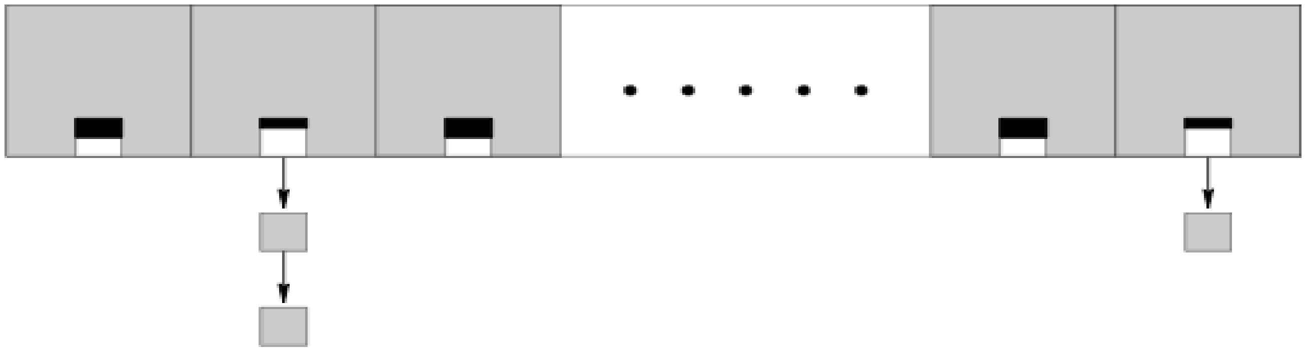
Method 3: Adaptive sorting [PPT04]

- If a relation is nearly sorted according to the join attribute, we wish to be able to exploit this.
- **Idea:** Merging almost-sorted sublists can be done efficiently if we may ”throw away” a few troublesome elements.
- If thrown-away elements fit in internal memory, we may compute the full join by one extra scan of the relations.

A better external hash table [JP06]

Ideas (some old, some new):

- Keep hash table blocks very close to full. (There will be many overflows.)
- Keep in each block a **buffer** of updates that need to be done in the overflow chain - almost all updates use 2 I/Os.
- Maintain records **ordered** such that almost all searches only need 1 I/O.



Hashing without a hash function?

- Problem with hashing: May fail to have good performance. Would like *deterministic* data structure with same performance.
- Most experts believe this is not possible on a ”normal computer”.
- But what if we have more resources - e.g., an a array of 64 disks that we can access in parallel? (Such storage systems exist.)

Deterministic load balancing [BHPPRT06]

Idea:

- Associate (in a clever way) with each possible key x a **fixed** set of blocks where it may be stored.
- One possible block on each disk - lookup in 1 parallel I/O.
- When inserting a key x, check all possible positions (in parallel) and insert x in the *least full* block.
- This **always** works if (roughly):
  - Blocks have room for *log N* keys.
  - There are *log(#possible keys)* disks.