A Basic Query Processor

Christoph Koch

Query Engines

- In this lecture we discuss the core of a database query engine, and some essential concepts.
- Key to ground the later material in the physical reality.
- Not just relevant for building databases!
- Precomputation: indexing.
- Query processing
  - Use algebraic plans = description of how data processing algorithms are to be composed.
- Query optimization: heuristic and cost-based.

Memory hierarchies and storage devices

- Memory hierarchies get more pronounced as time progresses.
- Several levels of processor caches, main memory, disk caches, disks, tapes; new storage devices.
- Hard disks
  - Sequential vs. random access
  - Latency vs. bandwidth
  - Disk parameters: seek time (to arbitrary/next track), transfer time; capacity; block size, sectors, tracks, surfaces.
  - Read and write times the same
- Flash memory
  - read, write, and erase times differ
  - finitely many writes/erases (but MTTF in disks bounded too)
  - can only erase very large blocks.
- Data streams arriving from a network.
- Unless stated otherwise: "secondary storage"=disks.

Pages and memory management

- Disks read/write data in minimum chunks (pages).
- Organize data sets in pages.
- Memory buffers: pages.
- Buffer manager swaps data between disk and main memory.
- Optimize secondary-storage processing for page-based processing.

Cost of secondary-storage algorithms: hard disks

- Seek time t_{seek} ($\approx$ 10ms): time to move the read/write head to the position to be read/written.
- Transfer time per page t_{transf_pg} ($\approx$ 0.1ms):
  - Average time to read one page if head is positioned.
  - Read/write sequence of n pages: time $t_{seek} + n \cdot t_{transf\_pg}$.

Cost of execution of secondary-storage algorithm:

- Cost in #page I/Os (read or write)
- Cost in #seeks
- Cost in seconds:

$$t_{transf\_pg} \cdot \#\text{page I/Os} + t_{seek} \cdot \#\text{seeks}.$$

- Assumes that I/O cost dominates CPU cost (true in most scenarios we will encounter).

The Relational Model

- Relations are sets (multisets) of tuples.

R	A	B
	1	2
	3	4
	3	4

- Schema $R(A, B)$; $sch(R) = \{A, B\}$. Column names (and their data types).
- Arity (number of columns) $ar(R) = 2$
- Cardinality (number of tuples in relation) $||R|| = 3$.
- $|R|$: number of pages used to store relation:

$$|R| := \left\lceil ||R|| \ / \ \underbrace{\left\lfloor \frac{\text{page size}}{\text{tuple size}} \right\rfloor}_{\text{\#tuples/page}} \right\rceil$$

Joins

- Input: Two collections C_1, C_2 of data items.
- Each collection stored sequentially on disk.
- Goal: find all pairs i_1, i_2 of items $i_1 \in C_1, i_2 \in C_2$ that match a given condition.

- Block I/O: Block nested loops (BNL) join.
- Exploiting previously sorted data: Index nested loops join, sort-merge join.
- Bucketization: (GRACE) hash join (multi-phase with two hash functions).

BNL Join $R \bowtie_\theta S$

Naive nested loops join:

for each tuple $r \in R$ do
  for each tuple $s \in S$ do
    if $\theta(r, s)$ is true then output (r, s)

BNL join, mem buffers B_R and B_S for pages from R and S, respectively. (Set $|B_S| = 1$ and maximize $|B_R|$.)

foreach block of $|B_R|$ consecutive pages of R on disk do
  seek and read the next $|B_R|$ pages of R into B_R
  seek the start of S
  foreach page of S do
    read the next page of S into B_S
    perform an in-memory join $B_R \bowtie_\theta B_S$; output result tuples

Creating cost functions

- Cost function: *coarsening* of the algorithm. Rather than compute the result, compute the number of instructions/IOs/seeks.
- There is no such thing as just one correct cost function for an operator. How precise do you want to be?
- Cost function must be cheap to evaluate (closed form, no looping). Tradeoff with precision.
- How many cases (based on data characteristics, resource availability) do you want to handle?

Cost of Join Operators: Example, BNL Join $R \bowtie_\theta S$

foreach block of $|B_R|$ consecutive pages of R on disk do
  seek and read the next $|B_R|$ pages of R into B_R
  seek the start of S
  foreach page of S do
    read the next page of S into B_S
    perform an in-memory join $B_R \bowtie_\theta B_S$

I/O cost (pages), output to pipeline (not to disk):

$$\underbrace{|R|}_{\text{read outer}} + \underbrace{\lceil |R| / |B_R| \rceil * |S|}_{\text{read inner}}$$

Seek cost (#seeks), output to pipeline (not to disk):

$$2 * \lceil |R| / |B_R| \rceil$$

(If the outer relation is read from the pipeline, the cost reduces to $\lceil |R| / |B_R| \rceil * |S|$ I/Os and $\lceil |R| / |B_R| \rceil$ seeks.)

Coarsening to #IOs, BNL Join $R \bowtie_\theta S$

$\overbrace{\text{foreach block of } |B_R| \text{ consecutive pages of } R \text{ on disk do}}^{\lceil |R|/|B_R| \rceil \ *}$

~~seek and~~ read the next $|B_R|$ pages of R into B_R $+|B_R|$

~~seek the start of S~~

$\overbrace{\text{foreach page of } S \text{ do}}^{|S| \ *}$

read the next page of S into B_S $+1$

~~perform an in-memory join $B_R \bowtie_\theta B_S$; output result tuples~~

$$\lceil |R|/|B_R| \rceil * (|B_R| + |S| * 1) = |R| + \lceil |R|/|B_R| \rceil * |S|$$

Coarsening to #seeks, BNL Join $R \bowtie_\theta S$

$$\overbrace{\text{foreach block of } |B_R| \text{ consecutive pages of } R \text{ on disk do}}^{\lceil |R|/|B_R| \rceil \;\; *}$$

seek ~~and read the next $|B_R|$ pages of R into B_R~~ $+1$

seek the start of S $+1$

~~foreach page of S do~~

~~read the next page of S into B_S~~

~~perform an in-memory join $B_R \bowtie_\theta B_S$~~

IdxNL Join $R \bowtie S$

Schema R(A,B), S(B,C); **clustered** index for S on B.
p is the number of unbuffered nodes on a path from the root to a leaf in the B^+-tree. Output is not written to disk.

foreach block of $|B_R|$ consecutive pages of R on disk do
  seek and read the next $|B_R|$ pages of R into B_R;
  foreach tuple r in B_R do
    look up r.B in index S by B:
      seek and read a path to a leaf in the B-tree
      read all $|\sigma_{B=r.B}(S)|$ pages that contain tuples
        matching the join condition; output results

Here, when we coarsen, the problem is to give a closed-form bound for $|\sigma_{B=r.B}(S)|$, which can be as bad as $|S|$ (but will be much better in typical cases).

$$\begin{aligned} \#\text{IOs:} &\quad |R| + ||R|| * (p + |S|) \\ \#\text{Seeks:} &\quad |R|/|B_R| + ||R|| * (p + 1) \end{aligned}$$

Cost of join operators

	#page I/Os	#seeks
BNL join	$\lvert R\rvert + \lceil \lvert R\rvert / b_R \rceil \cdot \lvert S\rvert$	$2 \cdot \lceil \lvert R\rvert / b_R \rceil$
Hash join	$3 \cdot (\lvert R\rvert + \lvert S\rvert)$	$2 \cdot (\lvert R\rvert + \lvert S\rvert + b)$
Merge join[1]	$\lvert R\rvert + \lvert S\rvert$	$\lvert R\rvert / b_R + \lvert S\rvert / b_S$
Index NL join	$\lvert R\rvert + \lVert R\rVert \cdot (p+1)$	$\lvert R\rvert / b_R + \lVert R\rVert \cdot (p+1)$

- For hash and merge-join, a 1:n relationship is assumed.
- For index NL join, it is assumed that the partners in S of each R tuple fit into one page.
- b_R, b_S: number of buffer pages allocated for holding data from R, S.
- b is the number of buffer pages available for hash buckets.
- The cost estimate is for an index NL join with a clustered B-tree index where the matching S tuples for each R tuple fit on one page; p is as before (say $p = 3$).

[1] Assumes data is sorted.

Cost of join operators, ctd.

Experiment: read both relations from disk, discard output.

	#page I/Os	#seeks
BNL join	$\lvert R\rvert + \lceil \lvert R\rvert / b_R \rceil \cdot \lvert S\rvert$	$2 \cdot \lceil \lvert R\rvert / b_R \rceil$
Index NL join	$\lvert R\rvert + \lvert\lvert R\rvert\rvert \cdot (p+1)$	$\lvert R\rvert / b_R + \lvert\lvert R\rvert\rvert \cdot (p+1)$

Cost estimates (20 buffer pages available: $b_R = 19$; 1024 bytes/page; 128 bytes/tuple for both R and S; $\lvert\lvert R\rvert\rvert + \lvert\lvert S\rvert\rvert = 10000$):

$\lvert\lvert R\rvert\rvert$	$\lvert\lvert S\rvert\rvert$	join op.	#page I/Os	#seeks	cost(msec)
2	9998	BNL join	1251	2	145.1
2	9998	IdxNL join	9	9	**90.9**
100	9900	BNL join	1251	2	**145.1**
100	9900	IdxNL join	413	401	4051.3
5000	5000	BNL join	21250	66	**2785.0**
5000	5000	IdxNL join	20625	20033	202392.5

SPC Queries #1

- **Selection $\sigma_\phi(R)$**
  Selection condition ϕ: Boolean combination of expressions $t \theta t'$ where
  - t, t': either a column name from $sch(R)$ or a constant value and
  - θ is one of $=, \neq, <$, and $\leq$.

  Result: Those tuples of R for which ϕ is true.
- **Projection $\pi_{\vec{A}}(R)$**
  Assumption: $\vec{A} \subseteq sch(R)$.
  Result: For each tuple $\vec{a}\vec{b}$ of R in which $\vec{a}$ are the values in the columns $\vec{A}$ and $\vec{b}$ are the values in the remaining columns of R, return $\vec{a}$. The result consists of distinct tuples, i.e., duplicate tuples are removed from the result.

SPC Queries #2

- Relational ("Cartesian", thus C) product $R \times S$
  Assumption: $sch(R) \cap sch(S) = \emptyset$.
  Result: The set of distinct tuples $\vec{r}\vec{s}$ obtained by concatenating tuples $\vec{r}$ from R with tuples $\vec{s}$ from S.
- Column renaming $\rho_{A_1 \ldots A_{ar(R)}}(R)$
  Assumption: $ar(R) = k$.
  Result: If the schema of R is $R(B_1, \ldots, B_{ar(R)})$, rename column B_i to A_i, for each $1 \leq i \leq ar(R)$.

Select-Project-Cartesian product (SPC) queries: relational algebra queries built using the operations σ, π, and $\times$ (and ρ).

- Theta-join: $R \bowtie_\theta S := \sigma_\theta(R \times S)$.

Other languages equivalent to SPC

- FO: $\exists x_1 \cdots \exists x_n \; R_1(\vec{x}_1) \wedge \cdots \wedge R_m(\vec{x}_m)$
- Datalog notation: single nonrecursive rule, e.g.

$$Q(x,z) \leftarrow E(x,y), E(y,z).$$

- Also known as conjunctive queries.
- SQL: select-from-where queries (without union, except, aggregations, negation, or disjunction).

Example. Schema R(A,B), S(B,C). SPC query:

$$\pi_{r.A,s.C}(\sigma_{r.B=s.B}(\rho_{r.A,r.B}(R) \times \rho_{s.B,s.C}(S)))$$

SQL:

SELECT DISTINCT r.A, s.C FROM R r, S s WHERE r.B=s.B;

From SQL to SPC

- select-from-where conceptual evaluation:

  SELECT <columns> FROM $R_1, \ldots, R_k$ WHERE <condition>;

  for each tuple t_1 from R_1 do

  $\ddots$

  for each tuple t_k from R_k do

  if <condition> is true on $(t_1, \ldots, t_k)$ then

  output <columns> of $(t_1, \ldots, t_k)$.

- Translation to SPC:

$$\pi_{<columns>}(\sigma_{<condition>}(R_1 \times \cdots \times R_k))$$

Relational query optimization

- One main motivation for special-purpose query languages (as compared to general-purpose programming languages): query optimization. (Speeding up query processing.)

Classical techniques:

- Algebraic query rewriting heuristics
  - e.g., pushing selections and projections down the algebra tree as far as possible.
- Cost-based query optimization
  - Choosing among several alternative equivalent query plans using cost functions for the operations in the query plan and statistics about the data.
- Different operator implementations
  - e.g., for joins: merge join, hash join, index nested loop join, ...
- Using index structures, clustering, materialized views, ...

Algebraic Rewriting

- Some algebraic laws:
  1. $R \bowtie_\theta S \equiv S \bowtie_\theta R$ (commutativity)
  2. $R \bowtie_\theta (S \bowtie_\phi T) \equiv (R \bowtie_\theta S) \bowtie_\phi T$ (associativity)
  3. $\sigma_{\phi \wedge \psi}(R) \equiv \sigma_\phi(\sigma_\psi(R))$
  4. $\sigma_\phi(R \bowtie_\theta S) \equiv (\sigma_\phi(R) \bowtie_\theta S)$ if ϕ only refers to columns of R
  5. $\pi_{\vec{A}}(R \bowtie_\theta S) \equiv \pi_{\vec{A}}(\pi_{(\vec{A} \cup \vec{B}) \cap sch(R)}(R) \bowtie_\theta S)$ where $\vec{B}$ are those columns that occur in θ.
- Heuristics for optimization.
  - Pushing selections down
  - Pushing projections down
  - Join reordering
- Goal: Reducing intermediate result sizes.

Heuristic optimization example

- Input query:

$$\sigma_{C.cname='CS101' \wedge S.sid=T.sid \wedge T.cid=C.cid}(S \times C \times T)$$

- Schema

$$S[tudent](sid, sname), C[ourse](cid, cname), T[aken](sid, cid)$$

- Some ways to push selections down (and choose join orders):

$$(S \times \sigma_{C.cname='CS101'}(C)) \bowtie_{S.sid=T.sid \wedge T.cid=C.cid} T$$

$$(S \bowtie_{S.sid=T.sid} T) \bowtie_{T.cid=C.cid} \sigma_{C.cname='CS101'}(C)$$

$$S \bowtie_{S.sid=T.sid} (T \bowtie_{T.cid=C.cid} \sigma_{C.cname='CS101'}(C))$$

- Heuristics: First solution is very bad. Third probably best.

Selectivities and estimating sizes of (intermediate) results

- Plan cost estimation: Selectivities and estimating result size.
- Definition: Selectivity $sel_\phi[R]$: estimate for size reduction

$$sel_\phi[R] :\approx \frac{||\sigma_\phi(R)||}{||R||}.$$

- Example: Selectivity for a theta-join $R \bowtie_\theta S$,

$$sel[R \bowtie_\theta S] := sel_\theta[R \times S] \approx \frac{||\sigma_\theta(R \times S)||}{||R \times S||} = \frac{||R \bowtie_\theta S||}{||R \times S||}.$$

- Given a selectivity estimate $sel_\phi[R]$, estimate result size $||\sigma_\phi(R)||$ as $sel_\phi[R] \cdot ||R||$.

Obtaining selectivities

How to estimate selectivities?

- From statistics about the database produced by the database system, offline (e.g. histograms).
- R and S in 1:n relationship (for each S tuple, there is exactly one R tuple); make a uniformity assumption:

$$||R \bowtie_\theta S|| = ||S||; \qquad sel(R \bowtie_\theta S) = 1/||R||.$$

- If the data in column A is categorical with k categories (e.g., 50 states), can estimate $sel_{A=c}[R] :\approx 1/k$.
- Equality selections on keys return ≤ 1 tuple:

$$sel_{K=c}(R) = 1/||R||.$$

More on query evaluation

- Query plans: Relational algebra trees with annotations.
- Annotations:
  - For each operator, choice of implementation.
  - For NL joins: which input relation is the outer loop?
  - Convention: left child of join is outer loop relation.
  - Indication of how buffer pages are assigned.
- Pipelining versus materialization. Selection, projection, BNL join, and Index NL join can be pipelined.
- Outer and inner loops in joins.

- Page size (excl. header): 1024 bytes
- Schema:
  Students S(sid, sname),
  Take T(sid,cid),
  Courses C(cid, cname)
- Query:
  select S.sname from S, T, C
  where S.sid=T.sid
  and T.cid=C.cid
  and C.cname='CS101';
- Memory buffers: 22 pages
- $||S|| = 16000$ tuples @ 4+60 bytes; $||T|| = 256000$ tuples @ 4+4 bytes; $||C|| = 1600$ tuples @ 4+60 bytes (A tiny database: 3.1 MB)
- Ultranaive evaluation:

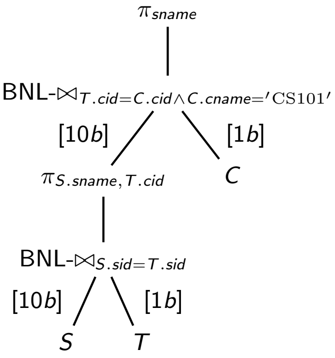
  for each tuple s of S on disk do
      for each tuple t of T on disk do
          for each tuple c of C on disk do
              if the condition on (s, t, c) holds then
                  output s.sname;

- Time cost: $(10 + 0.1) \cdot 16000 \cdot 256000 \cdot 1600$ msec $>$ 2000 years

Pipelined query evaluation

- Typical: pull interface. Each operator has a getNextResultTuple() call which steps through the algorithm until one new tuple is produced. Calls getNextResultTuple() of its input operator(s). Signals when no more tuples can be produced.
- Consequence: All operators run conceptually at the same time. Do not write result to disk but consume by subsequent operator.
- Costing subtleties. Example: BNLJoin.
  - Cost of scanning outer relation is ascribed to earlier operator.
  - I/O Cost: $\lceil |R|/b_R \rceil * |S|$.
  - Seeks for inner relation: 2 per scan of S. Possibly interrupts scan of another relation; need to jump back to where other operator was reading when finished.

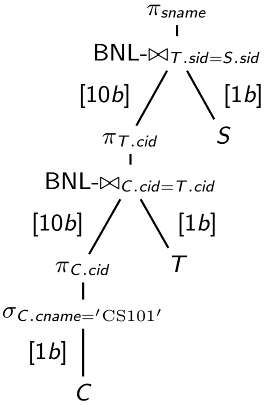
- Page size (excl. header): 1024 bytes
- Schema:
  Students S(sid, sname),
  Take T(sid,cid),
  Courses C(cid, cname)
- Query:
  select S.sname from S, T, C
  where S.sid=T.sid
  and T.cid=C.cid
  and C.cname='CS101';
- Memory buffers: 22 pages



Node	tp size	#tps/pg	#tps	#pgs	I/O pgs	#seeks
S	**4+60**	16	**16000**	1000	1000	100
T	**4+4**	128	**256000**	2000	-	-
$S \bowtie T$	72	14	256000	18286	1000/10*2000	100
$\pi(ST)$	64	16	256000	16000	0	0
C	**4+60**	16	**1600**	100	-	-
$ST \bowtie C$	128	8	160	20	16000/10*100	$2 * 1600$
(total)					361000	3400

Time cost: $\approx 10 \cdot 3400 + 0.1 \cdot 361000$ msec $= 70.1$ sec

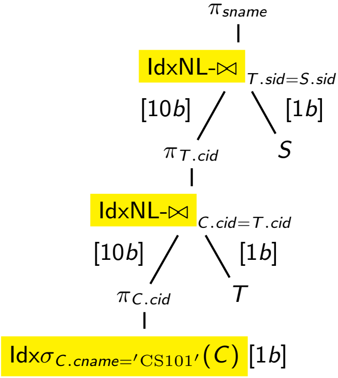
- Page size (excl. header): 1024 bytes
- Schema:
  Students S(sid, sname),
  Take T(sid,cid),
  Courses C(cid, cname)
- Query:
  select S.sname from S, T, C
  where S.sid=T.sid
  and T.cid=C.cid
  and C.cname='CS101';
- Memory buffers: 23 pages



Node	tp size	#tps/pg	#tps	#pgs	I/O pgs	#seeks
C	4+60	16	1600	100	100	1
$\pi(\sigma(C))$	4	256	1	1	0	0
T	4+4	128	256000	2000	-	-
$C \bowtie T$	12	85	160	2	2000	1
$\pi(CT)$	4	256	160	1	0	0
S	4+60	16	16000	1000	-	-
$CT \bowtie S$	68	15	160	11	1000	1
(total)					3100	3

Time cost: $10 \cdot 3 + 0.1 \cdot 3100$ msec $= 0.34$ sec

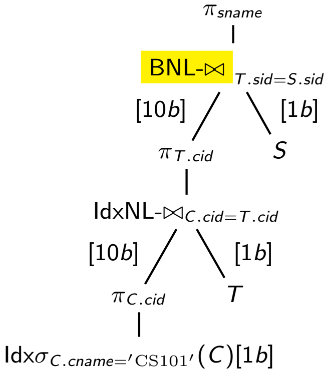
- Page size (excl. header): 1024 bytes
- Schema:
  Students S(sid, sname),
  Take T(sid,cid),
  Courses C(cid, cname)
- Query:
  select S.sname from S, T, C
  where S.sid=T.sid
  and T.cid=C.cid
  and C.cname='CS101';
- Memory buffers: 23 pages



Node	tp size	#tps/pg	#tps	#pgs	I/O pgs	#seeks
$Idx\sigma(C)$	4+60	16	1	1	3+1	4
$\pi(\sigma(C))$	4	256	1	1	0	0
T	4+4	128	256000	2000	-	-
$C \bowtie T$	12	85	160	2	3+2	5
$\pi(CT)$	4	256	160	1	0	0
S	4+60	16	16000	1000	-	-
$CT \bowtie S$	68	15	160	11	160*(3+1)	640
(total)					649	649

Time cost: $10 \cdot 649 + 0.1 \cdot 649$ msec ≈ 6.6 sec

- Page size (excl. header): 1024 bytes
- Schema:
  Students S(sid, sname),
  Take T(sid,cid),
  Courses C(cid, cname)
- Query:
  select S.sname from S, T, C
  where S.sid=T.sid
  and T.cid=C.cid
  and C.cname='CS101';
- Memory buffers: 23 pages



Node	tp size	#tps/pg	#tps	#pgs	I/O pgs	#seeks
$Idx\sigma(C)$	4+60	16	1	1	3+1	4
$\pi(\sigma(C))$	4	256	1	1	0	0
T	4+4	128	256000	2000	-	-
$C \bowtie T$	12	85	160	2	3+2	5
$\pi(CT)$	4	256	160	1	0	0
S	4+60	16	16000	1000	-	-
$CT \bowtie S$	68	15	160	11	1000	1
(total)					1009	10

Time cost: $10 \cdot 10 + 0.1 \cdot 1009$ msec ≈ 0.2 sec

Observations

- Indexes are not always worth using.
- Although NL join sounds naive, BNL join is often very good (and may beat hash joins and sort-merge joins).
- Choose IdxNL join over BNL join if there are very few tuples in the outer relation and the index is clustered.
- If unclustered, each tuple should have very few partners.
- Join order is usually even more important than choice of operator.
- Join order is more difficult to optimize than choices of operator implementations, because the former must be decided globally while the latter can be chosen individually, operator by operator.
- Often left-deep plans are quite good because they admit pipelining.

The System R algorithm

- We could examine all possible rewritings (up to a certain size) of a given query, estimate their cost, and choose the cheapest.
- But there are far too many query plans!
- Query optimization should save time, otherwise we could just use the query plan we start with!
- System R algorithm. Two ideas:
  - dynamic programming (bottom-up); i-th phase builds i-relation plans from $(i-1)$-relation plans;
  - considers only left-deep plans.
- The plans above were System R plans.

Bibliography

- Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems, 3rd Edition. Morgan Kaufmann, 2002.