Lecture 3 - IBM Papers

Access Path Selection (Selinger)

Intro, setup

- System R being developed since 1975; this paper is from 1979
- This is the bible of query optimization
- Remember, coming from the CODASYL era, so some of this is non-obvious:
  - A user does not need to provide join paths (remember nested loopy-approach?)
  - A user does not need to decide how each table is accessed
  - And the system figures out how to optimize to minimize cost
- What was known about query optimization at this point?
  - "Agile" First cut:
    - Divide and Conquer
      - Break complex expressions into subproblems and optimize those.
      - INGRES's *one-variable query processor* (DECOMP)
      - Strategy:
        - Pick the smallest table.
        - For each tuple, recursively evaluate the remainder of the query.
          - Essentially nested loops beginning with smallest relation as outer loop
          - Q: Why is reducing intermediate result sizes valuable?
            - Intermediate result sizes may actually be larger than output sizes
            - They end up being materialized multiple times
            - Small intermediates can be kept in memory, pipelined
    - Transformation Rules
      - Push filters before joins. Filters are cheap.
- Paper asks: *what if you could build an oracle?*
  - Moving query processing from programming to specification
  - There is a **finite** space of possible query plans.
  - Optimization becomes **search**.
    - Enumerate all plans.
    - Choose the one that minimizes an objective *in the given context*.
- Essentially divides QO into three main problems
  - Define the **search space** (with pruning heuristics).
  - Build a **noisy cost predictor**.
  - Design an **intelligent search algorithm**.
- These can be improved *independently* — and the community has been doing exactly that for ~50 years.
- The query optimization process in System R involves four steps, pretty reasonable:
  - parsing - also decomposition into query blocks (SFW)
    - mostly checking sytnax
  - optimization -
    - first looks up tables and columns and verifies existence -- all via the catalog
    - identifies possible access paths, statistics
    - then selects access paths (aka optimizes)
      - does so per query block, after determining which query block needs to execute first
      - per query block, permutations of joins and which join is selected
    - output in some IR they call
  - code generation

- maps each physical operator into code
- the code involves accesses to the RSS (relational storage system)
  - remember breakdown from lecture 1 on query processor vs. storage manager?

Storage system:

- maintains tables, provides access paths (indexes), maintains buffers, locking (CC) - only method of CC at the time, logging (durability)
- provides "tuples at a time" - essentially "volcano model"
- access paths:
  - sequential scan (called a segment scan in this paper), one page at a time, return all tuples of the relation
    - here, a segment comprising a set of pages can have more than one relation - but as storage has become cheap, it's easier to more clearly demarcate pages of one relation from another
  - index scan, impl. via B+trees, returns tuples from leaf nodes
    - clustered if data is sorted in index order.
    - Q: why can an index sometimes be worse than a sequential scan?
  - both accept SARGs - eliminating tuples before provided as a result to the next operator
    - implicitly doing predicate pushdown

Cost estimation:

- Overall cost = pages read (IO) + . tuples processed (Compute)
- Dominant cost in 1979 was IO
- Statistics include
  - Table:
    - Cardinality, # pages, (also fraction of a segment that is for a given table)
      - For indexed attributes: # of distinct values, size of index, among others
    - Q: What other stats would be helpful?
- Using these, a "selectivity" is assigned to each predicate
  - Some somewhat reasonable, eg uniform distribution, independence of predicates
  - some pretty funny 1/3, 1/4
  - but easy to see how this can go pretty wrong already Q: thoughts?
    - Skew, correlations,
- Good news: we don’t need exact costs — just **correct rankings**
- Card for an expression (aka the RSICARD in paper) = product of cardinalities of relations and the predicate selectivities
- Physical properties of interest to track: sorted order, called "interesting order"
  - orders for ORDER BY, GROUP BY, join attributes,
- Cost for various physical operators:
  - Clustered index lookup:
  - Unclustered index lookup:
  - Mostly textbook stuff!

Join Implementations and Order Selection:

- Nested loops
- Sort merge join
  - produces an interesting order AND benefits from interesting orders
- Note: hash joins had not been invented yet! 1980s
  - requires more memory

- n! orders
- But joining the n+1th relation to the output of the first n, is independent of "how" the first n is done: optimal substructure => bottom-up Dynamic Programming
  - also known as Bellman's *Principle of Optimality*
  - DP extended with context (interesting order)
- Essentially, for each subset of relations to be joined, and each interesting order
  - maintain the best result
  - plans x number of interesting orders
- Restrictions in paper:
  - Cartesian products as late as possible
    - Q: Why is this generally a good idea?
    - Q: When can this be bad?
      - Consider three relations, A (aid, x), B (bid, x), C (x, y, payload)
      - A, B - both 100 rows
      - C - massive (rows), for each given value of x and y, each .
        - but for a given (x, y), single tuple
      - Index on (x, y) for C
      - Query: Natural join
      - Approach 1: (A x B) C -> 10000 lookups into C
      - Approach 2: (A C) B -> even with index, intermediate result size of 100 x = .
      - Approach 3: (B C) A would be even worse
  - Left-deep join trees - no bushy or right-deep
    - Bushy would increase complexity to
  - ORDER BY and GROUP BY last
    - Q: Why a bad idea? GROUP BY in particular can reduce result size considerably

Nested queries

- Scalar subqueries (or even uncorrelated subqueries involving IN etc) are evaluated eagerly
  - `SELECT NAME FROM EMPLOYEE WHERE SALARY = (SELECT AVG(SALARY) FROM EMPLOYEE)`
- Correlated subquery example:
  - `SELECT NAME FROM EMPLOYEE X WHERE SALARY > (SELECT SALARY FROM EMPLOYEE WHERE EMPLOYEE-NUMBER= X.MANAGER)`
- "Giant waving of hands"
  - either evaluate per outer tuple; and
  - cache query results; and/or
  - sort intermediates to make "inner" query evaluation cheap
  - future work: rewrite into joins!

Takeaways

- This is a great example of top-down systems research:
  - Start with the *whole problem*.
  - Factor it carefully.
  - Then iterate on components
- Databases as a Cross-Cut of CS
  - This is a classic "AI" problem:
    - Search (DP)
    - Statistical prediction

    - Decades before “AI” was fashionable.
- Database research is a license to do **computer science**, not just “systems” or “theory”.
- Some problems stay useful *and open* for decades.

Granularity of Locks and Degrees of Consistency (1976)

- This is actually two papers; latter one somewhat flawed

Lock Granularity

Setup

- Key question: what do we lock?
    - Fine-grained locks:
        - High concurrency
        - High overhead
    - Coarse-grained locks:
        - Low overhead
        - Poor concurrency
- Extreme cases:
    - Fine-grained - every bit locked
    - Coarse-grained - entire database locked
- Neither works

Core idea: hierarchical locking

- Idea: lock *what you use*, but leave breadcrumbs.
    - Allows different transactions to do different "amounts" (and "kinds") of work
- Hierarchy example:
    - Database → Table → Block → Tuple
- Base case:
    - X - exclusive access to node and implicitly to all descendents
    - S - shared access to node and implicitly to all descendants
        - Allow multiple "readers" to proceed in parallel
        - One issue is whether the writers get starved - but can be handled by lock table recognizing starvation
- Paper: "In order to lock a subtree rooted at node **R** in **share** or **exclusive** mode, it is important to prevent share or exclusive locks on the ancestors of **R**, which would implicitly lock **R** and its descendants. Hence, a new access mode, intention mode (I), is introduced."
    - I locks tag ancestor nodes, indicating that X or S locking will be done at a finer level, and prevents any locks on ancestors that will conflict with that
- Procedure:
    - Walk top-down.
    - Request **intent locks**:
        - IS, IX, SIX
        - Intent locks do not implicitly do any locking - requestor has to explicitly request locks below
        - IS allows locking below in S or IS
        - IX allows locking below in X, S, SIX, IX, or IS
        - SIX allows locking below in X, SIX, or IX
            - (IS, S is not needed because it happens "for free")

- At the target level, request S or X.
  - for example, S is only granted if all ancestors are in IX or IS mode
- Do not recurse further → save overhead.
- Request locks root to leaf, release leaf to root

- Lock lattice:
  - IS < {S, IX} < SIX < X
    - SIX targets reading a subtree and writing a portion of it

	NL	IS	IX	S	SIX	X
NL	YES	YES	YES	YES	YES	YES
IS	YES	YES	YES	YES	YES	NO
IX	YES	YES	YES	NO	NO	NO
S	YES	YES	NO	YES	NO	NO
SIX	YES	YES	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO

Table 1. Compatibilities among access modes.

-
- (Ignore NL)
- Notice SIX column is conjunction of S and IX columns
- Extensions:
  - DAGs (e.g., Table → Index and File)
    - S lock is implicit if at least one parent has S (or X) implicit/explicit
    - X lock is implicit if all parents have X implicit/explicit
  - Dynamic lock graphs (key-range locking)
    - Historically interesting.
    - Practically painful.
- Other details about Scheduling and Conversions - mostly not very interesting
- Another interesting compatibility matrix (from Bernstein and Goodman, Chapter 2)

	Read	Write	Increment	Decrement
Read	y	n	n	n
Write	n	n	n	n
Increment	n	n	y	y
Decrement	n	n	y	y

-

Degrees of Consistency

- *Is this the paper that introduces transactions? YES! two-phase locking? YES!*
- *transactions*: "sequences of atomic actions are grouped to form transactions"
  - transactions are the units of consistency, i.e., they ensure databases stay "consistent" in the sense of respecting real-world properties
    - things can be inconsistent in between
  - transactions are atomically done or redone
  - transactions are also therefore also units of recovery - during failure, txns may be undone

- By definition if serial, then this consistency property is maintained; if concurrent, we need locking to ensure consistency
- This was one of the first papers to argue that you could *quantify and configure the trade-off* between concurrency and consistency, rather than treating “transaction isolation” as all-or-nothing.
  - Left in user control, because users know what level of consistency is needed
- Introduces degrees of consistency from the level of a transaction, and also their implementations in the form of locking, and properties of these degrees
  - One main issue is the tight coupling between degrees and lock-based implementations
- System R top-down recipe:
  - **Define semantics first**
    - Specify what a transaction is allowed to *observe* under different degrees of consistency.
    - What we would now call *isolation levels* (the “I” in ACID)
    - Or *degrees of consistency* (terminology that unfortunately stuck)
  - **Then define mechanisms**
    - Design locking protocols that enforce those semantics.
  - This separation of *what* from *how* is a recurring System R theme.

Introduction to the degrees of consistency

- A crucial distinction is drawn between:
  - **Dirty (non-atomic)** data: produced by transactions that may abort
  - **Committed** data: produced by transactions that have completed ("abdicated its right to undo")
- Degrees of consistency are defined in terms of which of these a transaction may observe.
- Degree 0 (ALLOWS DIRTY WRITES, but no DIRTY OVERWRITES) consistency: T
  - does not overwrite dirty data of other txns
- Degree 1 (READ UNCOMMITTED) consistency: Degree 0 +
  - does not commit any writes until EOT
- Degree 2 (READ COMMITTED) consistency: Degree 1 +
  - does not read dirty data of other txns
- Degree 3 (REPEATABLE READ) consistency: Degree 2+
  - other txns don't dirty any data read by T before T completes
- Recoverable - txns that can be undone, unrecoverable otherwise
  - Degree 0 - not recoverable because they write partial data out ("commit" partial data).
  - Degree 1 - if all other txns are at least degree 0, then degree 1 txn is recoverable
    - If all transactions see at least degree 1 then all are recoverable
  - Degree 2 - isolates txns from uncommitted data of other txns
  - Degree 3 means that same data read twice by txn won't be inconsistent

Lock Protocols

- Share, Exclusive modes as before
- Short duration - just one action, Long - until end of txns
- One definition:
  - T observes degree 0 (allows dirty write) if it sets a (possibly short) X lock on any data it dirties
    - these writes are "committed" immediately
  - T observes degree 1 (READ UNCOMMITTED) if it sets a long X lock on any data it dirties
  - T observes degree 2 (READ COMMITTED) if it sets a long X lock on any data it dirties, and also a (possibly short) S lock on any data it reads

    - T observes degree 3 (REPEATABLE READ) if it sets a long X lock on any data it dirties and a long S lock on any data it reads
- Improved definition:
    - ***well-formed***ness = locking in appropriate mode before doing an action
    - ***two phase*** = no locking after unlocking
    - degree 0 = well-formed wrt writes
    - degree 1 = well-formed wrt writes and two phase wrt writes (doesn't start writing stuff out until it is ready to commit)
    - degree 2 = well-formed overall and two phase wrt writes
    - degree 3 = well-formed and two phase overall

Schedules

- *Does this paper introduce schedules as well? YES! And serial schedules? YES!*
- History or audit trail of transactions
- *serial* = one transaction at a time
- any merging of actions of a set of transactions is called a schedule for the transaction
- *legal* = locking sequences in the schedule are allowed
- one can reason about "degree" within a schedule, as one can infer whether data read/written is dirty or not based on locking

Equivalences of Locking and Consistency

- If each txn observes a given lock protocol degree of consistency, then all txns "sees" that degree of consistency in that schedule.
    - Conversely, if the txns does not set locks in that way, then one can develop schedules with the corresponding violations
- Each transaction can choose its locking degree of consistency independent of others (and get those guarantees), as long as all transactions have degree 0

Dependencies

- Three dependencies edges in schedule:
    - <: W->W (same object written by two txns)
    - <<: W->W or W->R
    - <<<: W->W or W->R or R->W
- The schedule has a given degree of consistency 1, 2, 3 iff the edges with <, << or <<< resply form a partial order
- T1 (RA), T2 (RA), T2 (WA), T1 (WA)
    - T2 < T1, T2 << T1 and T2 <<< T1 based on the writes.
    - In addition, T1 <<< T2 based on the R->W edge.
    - So: acyclic degree 2 but not degree 3.
    - T1 sees "degree 2" but T2 sees "degree 3"

Beautiful summary of the paper

ISSUE	DEGREE 0	DEGREE 1	DEGREE 2	DEGREE 3
COMMITTED DATA	WRITES ARE COMMITTED IMMEDIATELY	WRITES ARE COMMITTED AT EOT	SAME	SAME
DIRTY DATA	YOU DON'T UPDATE DIRTY DATA	0 AND NO ONE ELSE UPDATES YOUR DIRTY DATA	0, 1 AND YOU DON'T READ DIRTY DATA	0,1,2 AND NO ONE ELSE DIRTIES DATA YOU READ
LOCK PROTOCOL	SET SHORT EXCL. LOCKS ON ANY DATA YOU WRITE	SET LONG EXCL. LOCKS ON ANY DATA YOU WRITE	1 AND SET SHORT SHARE LOCKS ON ANY DATA YOU READ	1 AND SET LONG SHARE LOCKS ON ANY DATA YOU READ
TRANSACTION STRUCTURE see [1]	WELL FORMED WRT WRITES	(WELL FORMED AND 2 PHASE) WRT WRITES	WELL FORMED (AND 2 PHASE WRT WRITES	WELL FORMED AND TWO PHASE
CONCURRENCY	GREATEST: ONLY WAIT FOR SHORT WRITE LOCKS	GREAT: ONLY WAIT FOR WRITE LOCKS	MEDIUM: ALSO WAIT FOR READ LOCKS	LOWEST: ANY DATA TOUCHED IS LOCKED TO EOT
OVERHEAD	LEAST: ONLY SET SHORT WRITE LOCKS	SMALL: ONLY SET WRITE LOCKS	MEDIUM: SET BOTH KINDS OF LOCKS BUT NEED NOT STORE SHORT LOCKS	HIGHEST: SET AND STORE BOTH KINDS OF LOCKS
TRANSACT-ION BACKUP	CAN NOT UNDO WITHOUT CASCADING TO OTHERS	UN-DO INCOMPLETE TRANSACTIONS IN ANY ORDER	SAME	SAME
PROTECTION PROVIDED	LETS OTHERS RUN HIGHER CONSISTENCY	0 AND CAN'T LOSE WRITES	0, 1 AND CAN'T READ BAD DATA ITEMS	0, 1 ,2 AND CAN'T READ BAD DATA RELATIONSHIPS
SYSTEM RECOVERY TECHNIQUE	APPLY LOG IN ORDER OF ARRIVAL	APPLY LOG IN < ORDER	SAME AS 1: BUT RESULT IS SAME AS SOME SCHEDULE	RERUN TRANSACTIONS IN <<< ORDER
DEPENDENCIES	NONE	W→W	W→W W→R	W→W W→R R→W
ORDERING	NONE	< IS AN ORDERING OF THE TRANS-ACTIONS	<< IS AN ORDERING OF THE TRANS-ACTIONS	<<< IS AN ORDERING OF THE TRANS ACTIONS

What's missing?

- First, note **Degree 3 is NOT serializable**
  - Phantoms are not accounted for

    - **Snapshot isolation doesn't fit cleanly.** SI is widely used in modern MVCC systems, but it's *between Degree 2 and 3*: stronger than repeatable read for row values, weaker than serializable (allows write skew).
- Tight coupling between locking mechanism and the isolation level - hard to transfer over to other (non-logging-based) protocols

Historical Footnote

Pull-quote from the paper:

> "We wish to emphasize that this system is a vehicle for research in database architecture, and does not indicate plans for future IBM products."

Takeaways for Modern Researchers

- Degrees of consistency and transactions were *not* handed down by theory; they were co-designed with real systems.
- The System R group solved semantics *and* performance problems simultaneously.
- Locking overhead and usability were first-class concerns from the beginning.