DB101 – Query processing

Goetz Graefe – Madison, Wis.

Agenda

- Introduction – compilation vs interpretation

Query execution

- Scans of rows & columns
- Duplicate removal & grouping & pivoting
- Join types & algorithms
- Plans using indexes
- Parallel query execution

Query optimization

- “Access plans” incl joins
- Cost calculations
- Cardinality estimation
- Dynamic, robust, etc.
- Distributed, semi-joins
- Testing & verification

- Summary and conclusions

Topics omitted

1. Update plans for indexes, views, constraints, triggers
2. Automatic index creation & removal
3. Automatic statistics creation & refresh
4. Semantic query optimization & rewrite
5. Hardware support: ISA, FPGA, GPU
6. Optimizer software updates, robust performance
7. XQuery/JSON translation & full-text search
8. Rewriting user-defined functions

Query execution – overview

- Logical vs physical algebra
- Plan caching, plan validation
- Scans
- Index, sort-, and hash-based matching algorithms
  - Interesting orderings
- “Access plans” vs dataflow plans
- Serial and parallel query execution
  - Co-partitioning, interesting partitioning

Query execution – engine architecture

Physical (execution) algebra

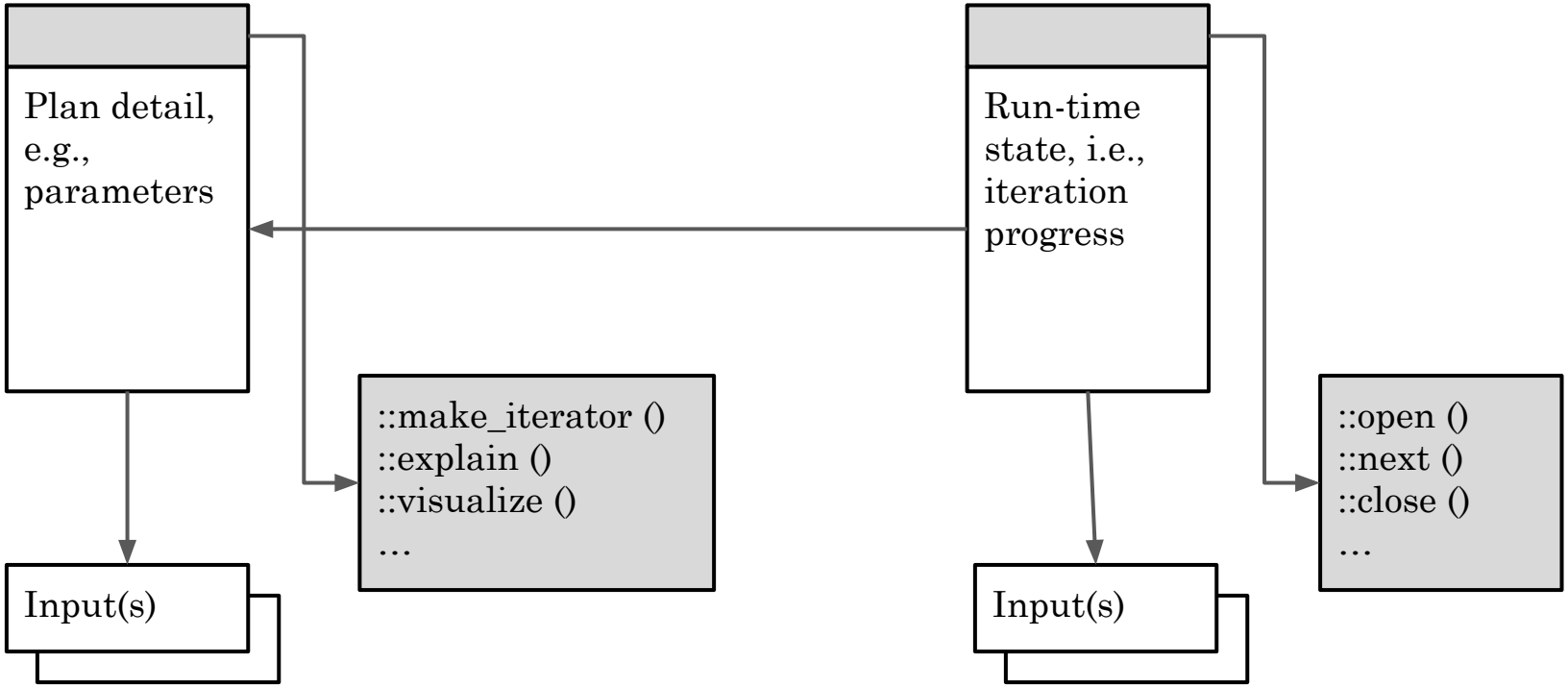
- Scans & iterators
- Unary matching: “distinct”, grouping, pivot, compression
- Binary: (inner, outer, semi) join, set ops (intersect etc.)
  - Bit vector (Bloom) filters
- Nested iteration
- Parallel query execution: activation & scheduling
- Memory management, scheduling bushy plans

Query execution – scans

- Per storage format: open(), next(), close() methods
  - Row & column stores, indexes, …
- Read-ahead, buffer pool pollution
- Index- vs storage-order scans
- Merging scan (e.g., in LSM-forests)
- Shared scans
- Opportunistic scan (buffer pool)
- Sampling scans

Query execution – plan & iterator nodes

- Traditional methods: open() – next () – close ()



Query execution – iterator methods

- Traditional: open() – next () – close ()
- Segmented execution:
- Nested iteration: rewind () – rebind ()
- Parallel execution: early_phases () – local_open/_close ()
- Distributed execution: global_open () – global_close ()
- … pause () – resume ()

Query execution – iterators

- Scans, sort, filter, (project)
- Order-, index-, hash-based unary & binary matching
- “printf”, “assert”
- Verification (sort order, partitioning, …)
- Exchange/shuffle (parallelism)
- Spool (shared intermediate results)

Modifying an existing sort order

- Segmented sorting
  e.g., case 1
- Merging pre-existing runs
  e.g., case 2

Case	Sort order	
	from	to
0	A,B	A
1	A	A,B
2	A,B	B
3	A,B	B,A
4	A,B,C	A,C,B
5	A,B,C	B,A,C
6	A,B,C,D	A,C,B,D

Query execution – unary matching

- ‘Distinct’, ‘group by’, ‘pivot’, compression
  Expensive actions: fetch, look-up join, nested iteration, user-defined functions
- Order-based algorithms
  - In-stream grouping (rollup?)
  - In-sort grouping
- Index-based grouping
- Hash-based grouping: “hash-aggregation”

Abstract interfaces: row formats & methods

Query execution – binary matching

- Joins (inner, outer, semi, mark)
  Set operations (intersect, except, union)
- Sort-based algorithms: merge join
- Index-based algorithms: index nested loops join
- Hash-based algorithms: hash join
  - Adaptive join
  - Symmetric (early output)
- “Co-group-by”

SQL Server – Merge join

"If the two join inputs are not small but are sorted on their join column (for example, if [...] scanning sorted indexes), a merge join is the fastest join operation.
If both join inputs are large and the two inputs are of similar sizes, a merge join with prior sorting and a hash join offer similar performance.
However, hash join operations are often much faster if the two input sizes differ significantly from each other."

SQL Server – Recursive hash join

"If the build input is so large that inputs for a standard external merge [sort] would require multiple merge levels, multiple partitioning steps and multiple partitioning levels are required.
If only some of the partitions are large, additional partitioning steps are used for only those specific partitions.
In order to make all partitioning steps as fast as possible, large, asynchronous I/O operations are used so that a single thread can keep multiple disk drives busy."

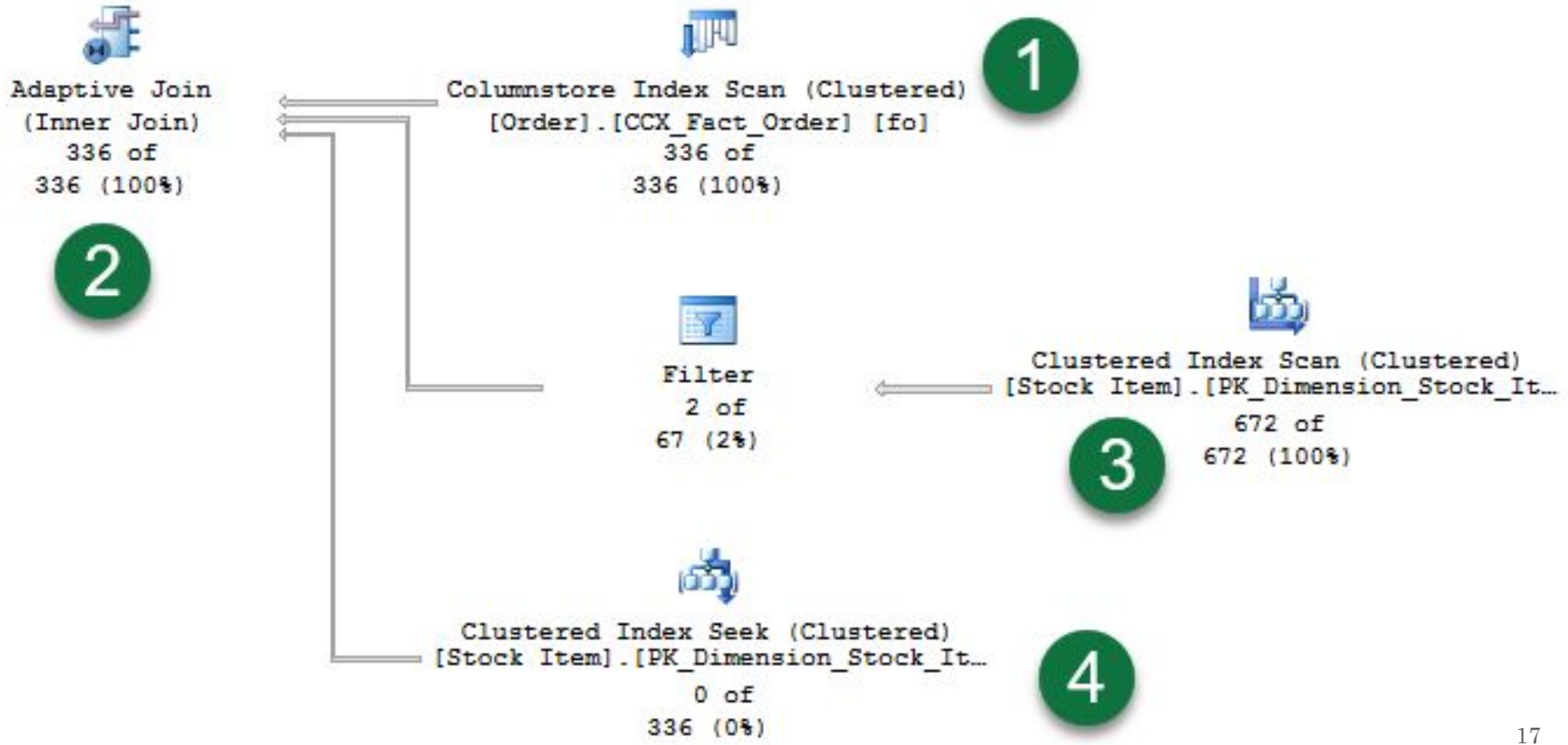
SQL Server – Hybrid hash join

"If the build input is only slightly larger than the available memory, elements of in-memory hash join and grace hash join are combined in a single step, producing a hybrid hash join.

It is not always possible during optimization to determine which hash join is used. Therefore, SQL Server starts by using an in-memory hash join and gradually transitions to grace hash join, and recursive hash join, depending on the size of the build input."

highlights of SQL Server 7 [1998]

SQL Server – Adaptive join



Dynamic join sequence – lookup joins

- Outer-most ordered “driving” scan
  + linear sequence of lookup joins
- “Bubblesort” of join operations (by selectivity)

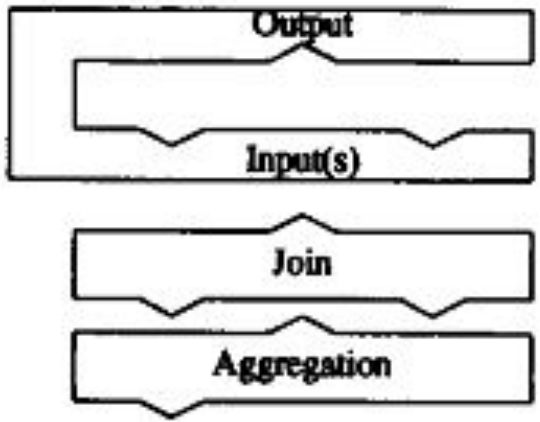
Query execution – index plans

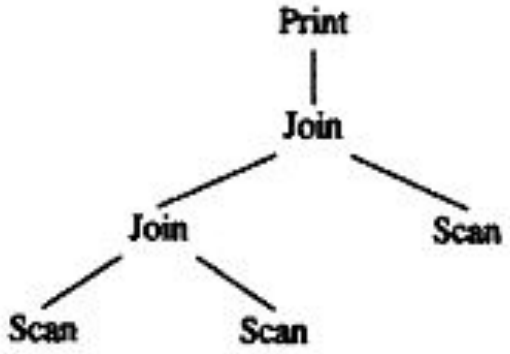
- Fetching rows – lookup join
  - Batches, asynchronous I/O
  - Sorting
- MDAM search & merge: “Y=5” on index (X, Y, Z)
- Index intersection, union, difference, e.g.,
  “T.A in (3, 5, 7) and T.B between 1 and 8 and T.C <> 64”
- “Covering indexes” for “index-only retrieval”
  - Index-to-index nav’n, e.g., student-enrollment-course
  - Index joins, e.g., “select A, B from T where…”

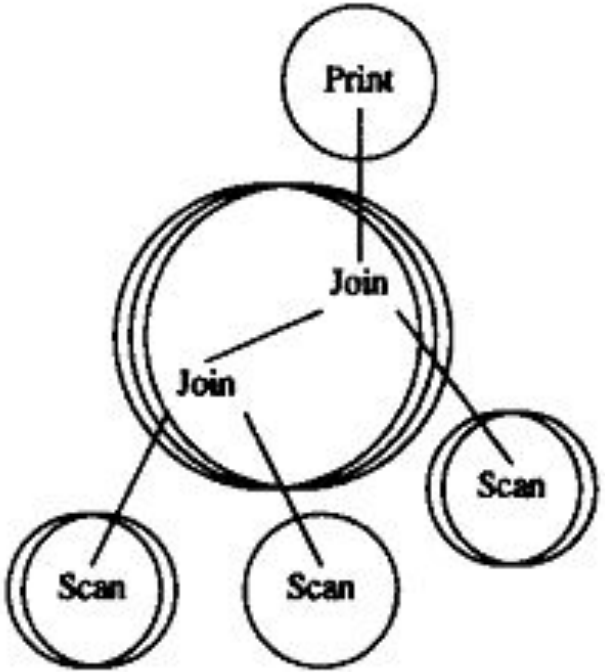
Query execution – parallel execution: “exchange”

- Exchange ≅ shuffle
  - Database query execution vs data analytics
  - In-pipeline vs pipeline breaker
  - Order-destroying vs order-creating
  - Hash join vs “reduce”
  - Hash-partitioning vs range-partitioning
- “Interchange”

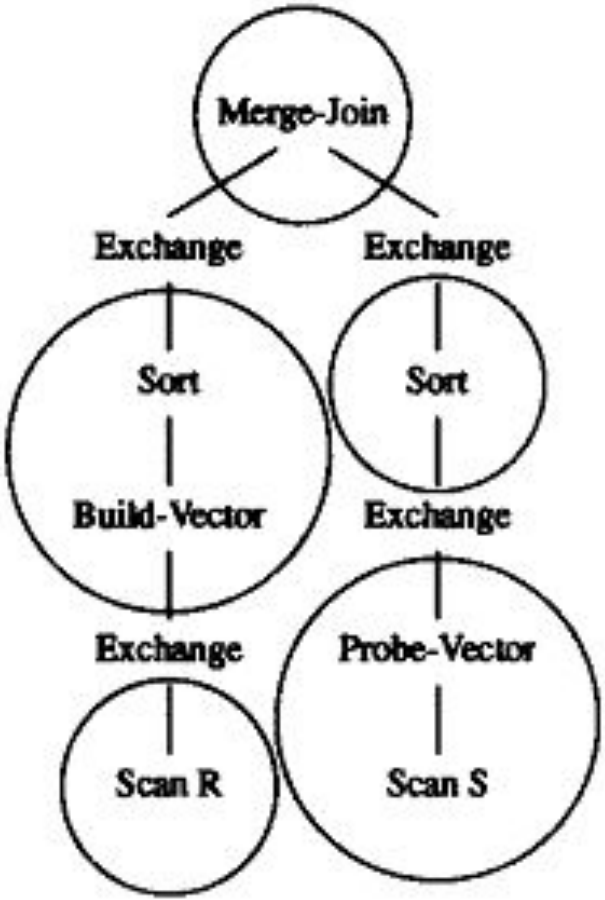
Parallel query plans in Gamma and in Volcano







Parallel merge join with bit vector filtering



Query optimization

- SQL text, AST, join graph,
  logical (relational) algebra, physical (execution) algebra
- Metadata, catalogs – name disambiguation
  - Logical & physical properties
- Cardinality estimation, cost calculation
- Plan space exploration, plan selection
  - Required & delivered & convenient physical properties
- Plan compilation, plan caching
  - Plan parameters, security context

Cost models & metrics

- CPU + I/O [1979]
- Network
- Memory usage?
- Row count? (only sort? +UDFs?)
- Robustness? (slope + jumps)

Cardinality estimation

- Table sizes + “magic numbers”, e.g., “=”: 10%, “<”: 45%, …
- Histograms
  - Equi-width
  - Equi-area (~~equi-height~~) ⇔ quantiles (b-tree root?)
  - Most frequent values ⇐ join size errors
  - Max-diff histograms (“=” for keys, “<” for measures)
- Sketches, wavelets, polynomials…
- Machine learning…

Cardinality estimation: selectivity errors

- Correlations ⇐ timestamps, dates, money
  - e.g., “line_items.commit_date < line_items.ship_date”, “line_items.ship_date < orders.order_date”
  - Cross-tab, difference, factor
- Functional dependencies
  e.g., “make = ‘Honda’ and model = ‘Accord’ ”
- Trends and patterns, e.g., seasonal sales
- Pattern matching “like”

Cardinality estimation: join size errors

- Functional dependencies
  e.g., “t1.make = t2.make and t1.model = t2.model”
- Referential integrity = foreign key integrity constraints
  e.g., “l.order# = o.order#”
- Frequent values, duplicate output rows
  e.g., “orders.customer# = invoices.customer#”

Logical & physical properties (of intermediate results)

	Accurate	Estimated
Logical properties	Set of columns Integrity constraints	Row count Value distribution
Physical properties	Sort order Partitioning Encoding	Compression ratio

Property enforcers: sort, shuffle/exchange, …

Further candidates: columns in memory, Halloween, …

Plan exploration & selection

- Dynamic programming – System R [1979], Starburst [‘89]
  - Linear joins, interesting orderings: “access plans”
  - No grouping, no outer joins, no semi-joins
  - Very efficient plan selection & construction
- Incremental transformations – Exodus [‘87], Cascades [‘95]
  - Logical & physical algebras & properties
  - Transformation & implementation rules
  - Generality & extensibility, e.g., FFT placement

Query processing – summary

- Cost-based plan optimizations are required
- Cardinality estimation will never be precise
- Compile-time planning is not enough
- Run-time adaptive plans during query execution
    - Scans & data access
    - Join algorithms & join sequence
    - Parallelism & skew
    - Updates & index maintenance

highlights: important techniques
in query optimization and execution