Adaptive and Self-Tuning Query Processing

Ioana Manolescu
DEI, Politecnico di Milano

(soon: INRIA Futurs, Gemo team)

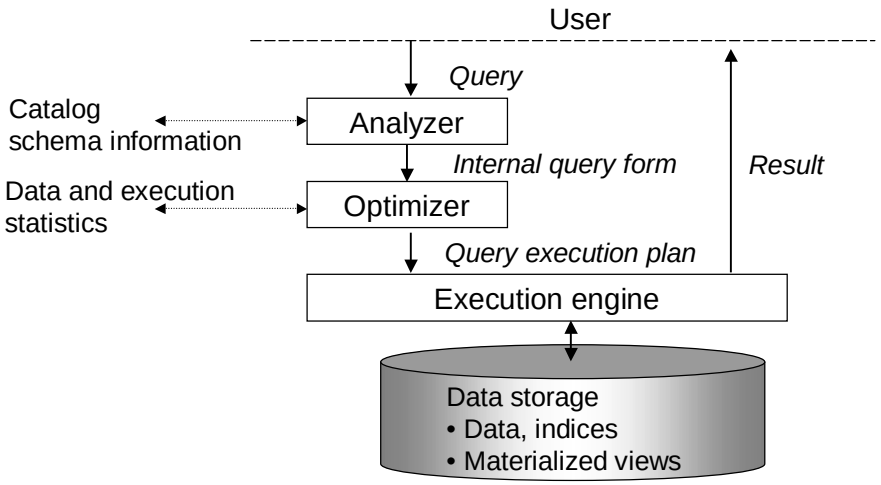
Plan

- Part 1: overview of traditional query processing
  - Query optimization
  - Query execution
- Part 2: adaptive and self-tuning query processing
  - The need for adaptativity
  - Existing solutions
  - Perspectives

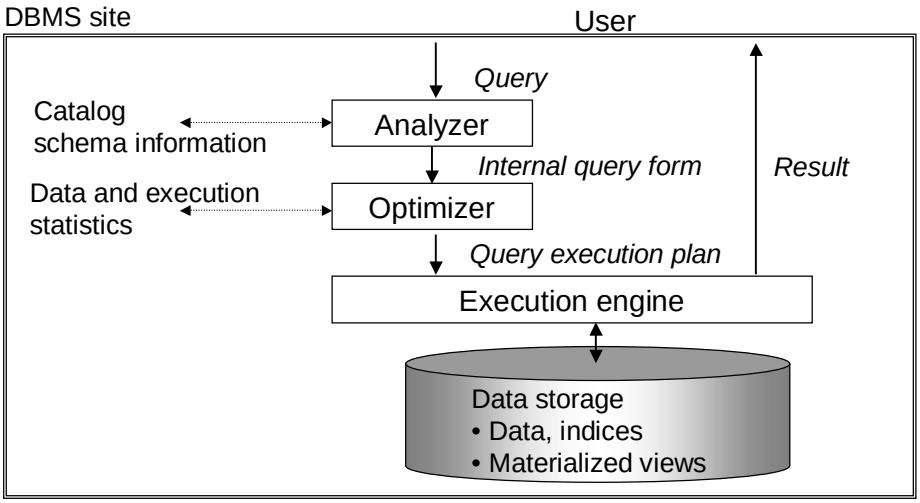
Part 1
Overview of traditional query processing

- General architecture of a query processor
- Brief overview of
  - Data storage
  - Query execution
  - Query optimization
- Going global: distributed query processing
  - Distributed DBMS
  - Wrapper-mediator systems

Generic query processor architecture



Centralized query processing (DBMS)



Data storage

- Data, indexes, materialized views, statistics
- Index: redundant structure supporting fast access to records in a table according to a search criterion
  - An index on R.a supports “select * from R where R.a=5”
- Materialized view: persistent result of a query
  - A materialized view “select * from R, S where R.a=S.b” can be used to answer “select R.c from R, S where R.a=S.b”
- Statistic: summary information on table or view column(s)
  - No. distinct R.a values, frequency of each R.a value
  - Implied by the presence of an index

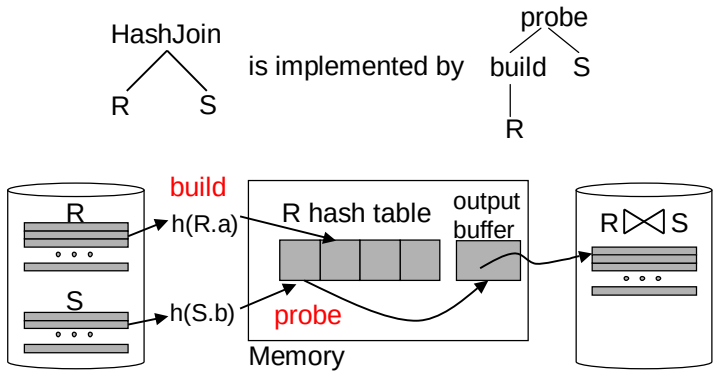
Query execution

- The execution engine contains a library of physical operators
  - *Scan(R)*
  - *B+ tree index lookup(R, X)*: using the B+ tree index on R.a, return R tuples such that R.a = X
  - *Index nested loop join(R,S, R.a=S.b)*:
    - foreach t in S
      - access matching tuples in R using R's index
        - foreach matching R tuple produce one output tuple
  - *Pattern Scan operator(XMLDoc, patt)*: using the XyIndex on XMLDoc, retrieve nodes matching *patt* [ABC2001]

Physical operators

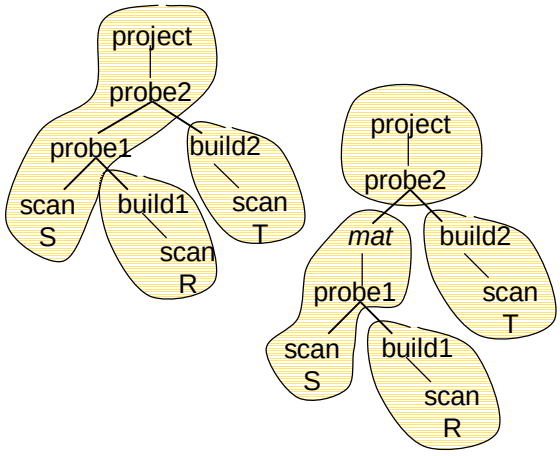
- Algorithms for implementing logical operators
- Run in memory and/or on disk
- Cost: disk I/Os
  - cost of IndexNLJ(S,R): read S + N_S * access R index
  - cost of HashJoin(R,S): read R + read S
- Before execution, memory is reserved for operators
  - For HashJoin(R,S), construct a hash table for R
- The memory needs depend on data statistics
  - Hash table for R: depends on R's size
    - R may be the result of a complex plan

Example: physical operators for HashJoin



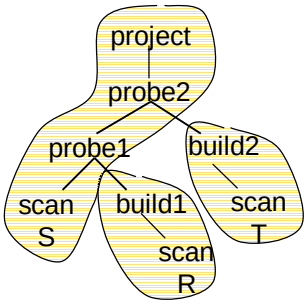
Execution of physical plans

- Producer-consumer dependencies among operators
- Data is
  - *materialized*
  - *passed along pipeline chains*
- Pipeline chains are delimited by blocking operators (build, mat, sort...)



Execution of physical plans

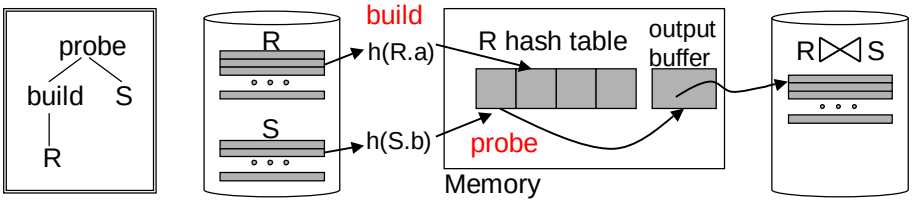
- Operators in a pipeline chain run together at a given moment
  - They have to fit in memory together
- Scheduling: order of execution of physical operators
  - scanR, build1, scanT, build2, scanS, probe1, probe2, project
- Memory allocation: splitting memory among operators running at the same time
  - scanS 10%, probe1 45%, probe2 45%, project 10%



Implementing physical operators

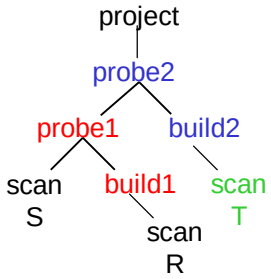
- The *iterator* model [Gra93]
  - Uniform, general interface for *any* physical operator
  - Three methods:
    - Open() sets up space, performs initialization
    - Next() returns one result tuple (or *eof*)
    - Close() releases resources and exits
- Data flows upwards, control flows downwards

Iterator example: HashJoin



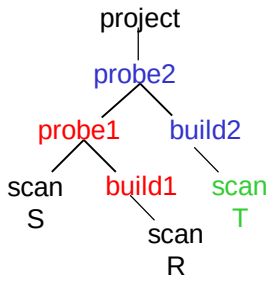
- probe.open(){
  build.open(); t = build.next()
  while (t != eof) {
  put t in table; t=build.next() }
  build.close()
  probe.open() }
- probe.next() {
  read t from S;
  probe the hash table with t;
  return one result tuple}
- probe.close() { de-allocate table }

Executing plans of iterators (1/3)



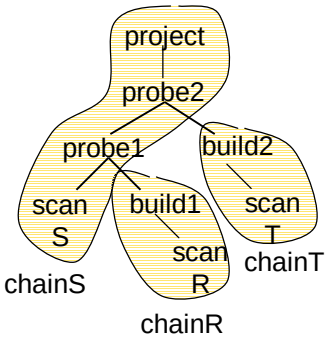
- project.open(){
  probe2.open{
  build2.open() { // build hash table for T
  scanT.open();
  while (! eof) {
  t = scanT.next(); insert t in table;
  } scanT.close();
  build2.close();
  probe1.open() {
  build1.open();
  ... // build hash table for R
  build1.close();
  } } }

Executing plans of iterators (2/3)

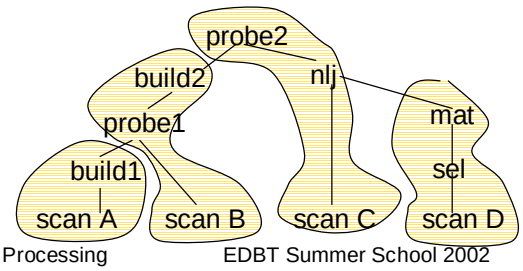


- project.next(){
  probe2.next {
  probe1.next() {
  s = next tuple from S;
  probe table for R with s;
  return a tuple rs;
  }
  probe table for T with rs;
  return a tuple rst;
  }
  rstp=projection(rst);
  return rstp
  }

Executing plans of iterators (3/3)

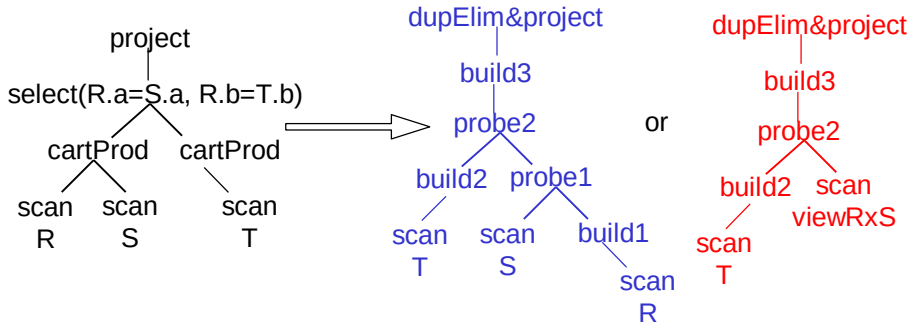


- Producer-consumer operator relationships induce a partial order among the executions of pipeline chains *(scheduling constraints)*
- The iterator implementation of physical operators completely determines the order of execution of the plan *(the scheduling)*
- Ex. scheduling for



Query optimization

- Input: a query in a machine-readable format
- Output: a physical query execution plan



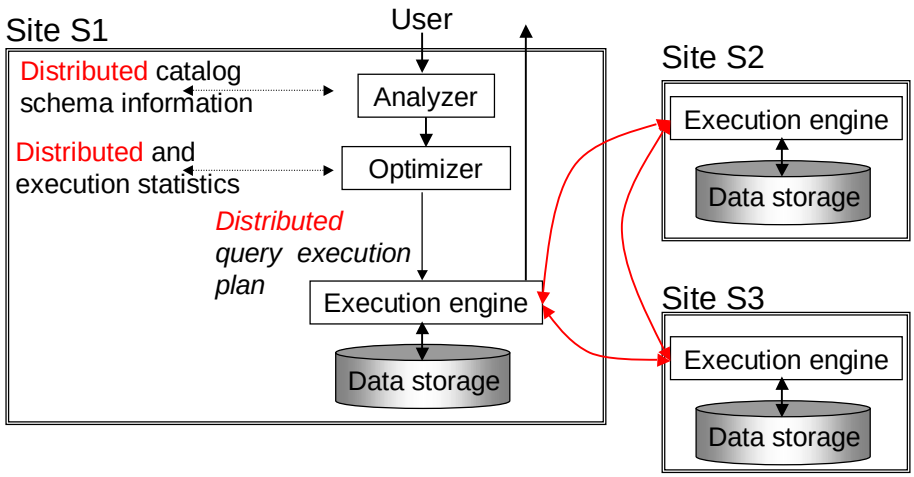
Query optimization: a search problem

- The space of physical plans *logically equivalent* to the optimizer's input: search space
- Every physical plan has a set of *properties*
  - Total work, e.g. total number of disk I/Os
  - Time to the first tuple
  - Time to the last tuple (to completion, response time)
- These properties are aggregated into a cost
- Optimizing = exploring *part of* the search space following a search strategy
  - returns an explored physical plan minimizing the cost

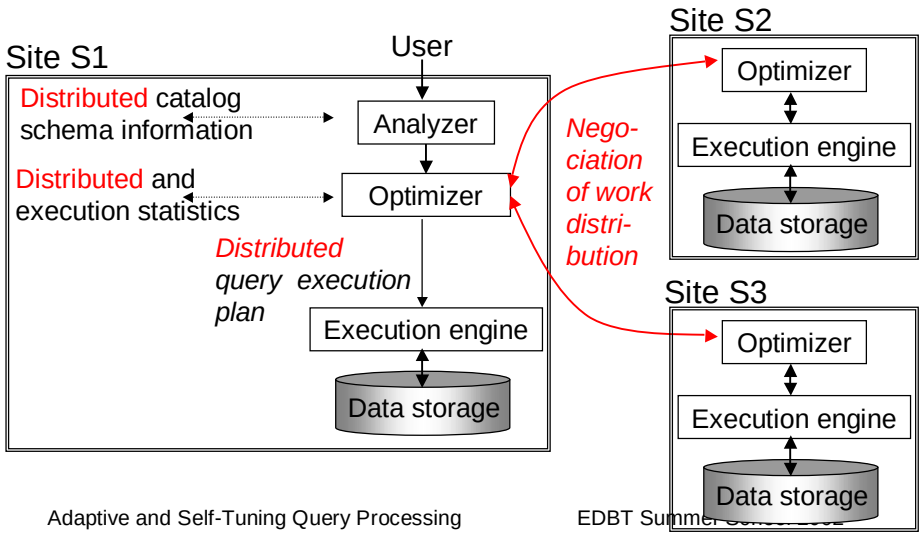
Distributed query processing: what changes

- Distributed query processing architectures
  - Distributed DBMS
    - Master-slave scenario
    - Negociation scenario
  - Wrapper-mediator systems
- Impact of distribution on
  - Query execution
  - Query optimization

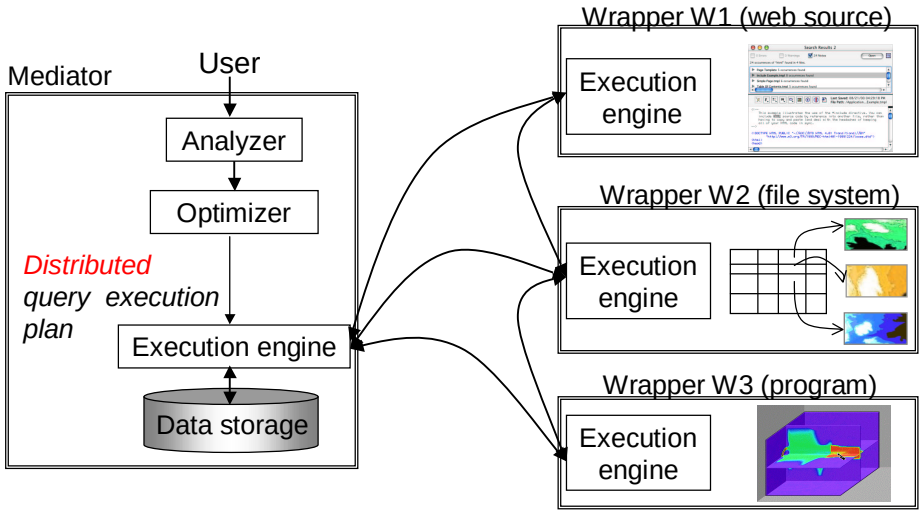
Distributed query processing system – master-slave scenario



Distributed query processing system - negociation scenario

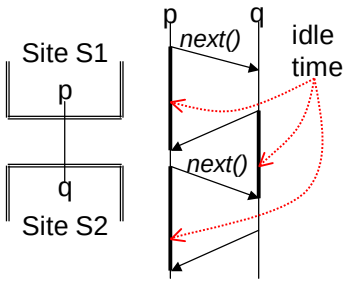


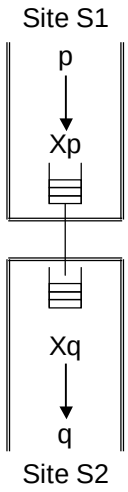
Wrapper-mediator system



Influence of distribution on query execution (1/2)

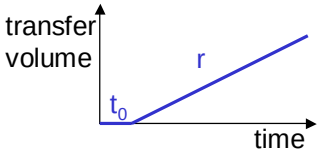
- Operators run on several sites
  - Opportunities for parallelism
  - The basic iterator model is sequential
  - Solution: Exchange operators [Gra93]





Influence of distribution on query execution (2/2)

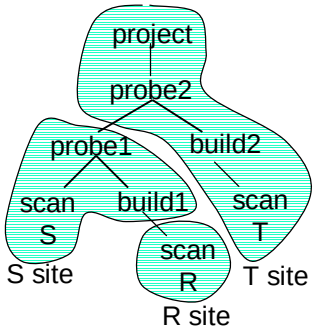
- Operators run on several sites
  - Variable performances of an operator on different sites
    - Different algorithms, memory conditions
  - Distributed scheduling
  - Remote sites may fail
- Intermediate results are transferred
  - Characterizing data transfer



t_0 startup cost
r transfer rate (bytes/sec)

Influence of distribution on query optimization

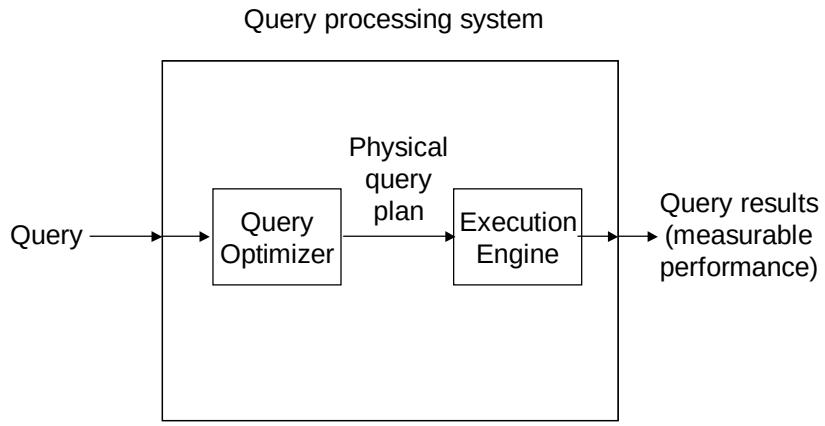
- The search space increases
  - Several sites to which an operator can be assigned
  - ... therefore, even stronger usage of heuristics
- The cost model incorporates
  - Data transfer times
  - Parallelism
    - Between processors
    - Between processor and transfer
  - Metric: response time
    - Until the last result tuple arrives at the query site



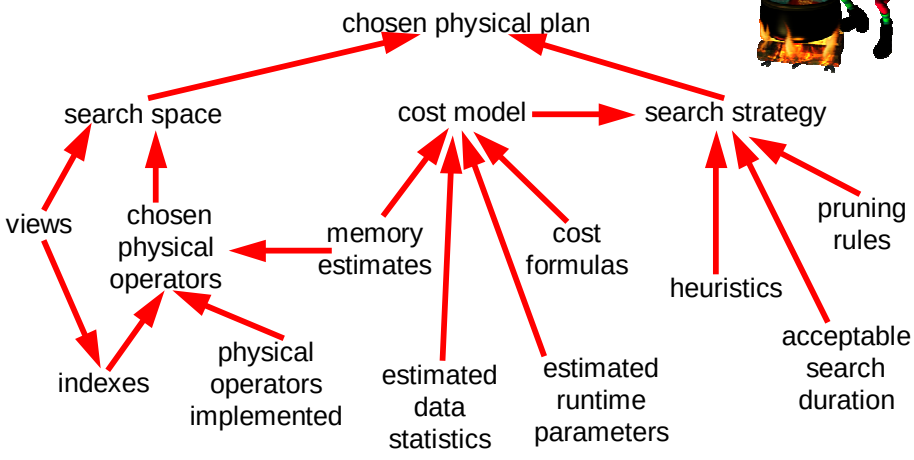
Query processing in wrapper-mediator systems

- Wrappers may provide only simple operators
  - Scan, callProgram(arg_1, arg_2, ..., arg_n)
- Statistics are absent or ill-defined
  - Data changes without notice
  - Data presented by wrappers can be the result of a mapping
    - "Average result of program *p*" ?...
    - Most frequent value of *//book/@title*, if the data is stored in relations ?...
- Loss of distributed view and control
  - Wrappers run on autonomous sites
  - We only control the mediator

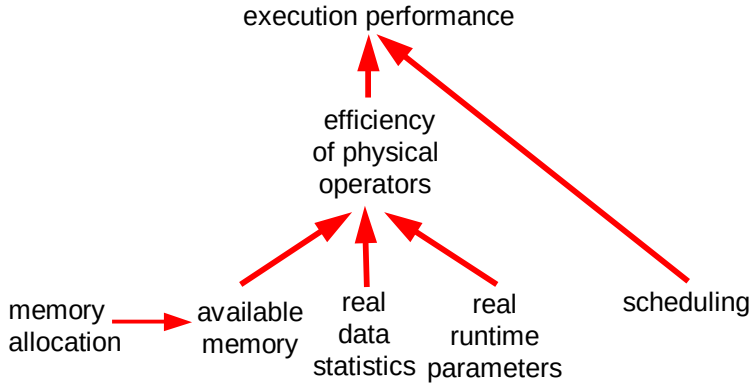
Putting it all together: what determines the performance of query processing ?



What determines the optimizer's choice



What determines the execution performance



Part 2: adaptive and self-tuning query processing

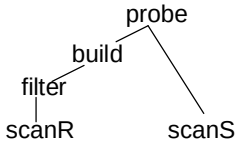
- The need for adaptativity
- Existing solutions
- Perspectives

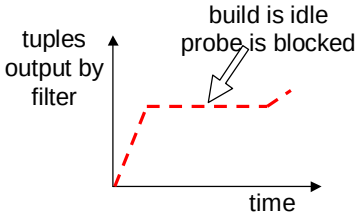
Adaptive query processing: definition

- An adaptive query processor [HFC+00]:
  1. Receives information from its environment
  2. Uses this information to determine its behavior
  3. Performs the two above in a feedback loop
- We adopt a broader perspective: adapting means
  1. Reacting to the unexpected (aka *dynamic)*
  2. Learning about the unknown (aka *self-tuning)*
     - For maintaining or improving query processing performance

Unexpected events at runtime (1/2)

1. Insufficient memory for an operator
2. Data transfer rates (distributed setting)
3. Data characteristics
   - Cardinality, number of distinct values
   - Value distribution (skew)
   - Order w.r. operators: “varying operator selectivity”





Unexpected events at runtime (2/2)

- Causes
  - Wrong estimates
    - Cardinality
  - Intrinsic variable parameters
    - Memory on a remote site, network bandwidth
- Occurence increasingly likely in
  - Centralized DBMSs
  - Distributed DBMSs
  - Wrapper-mediator systems
  - Changing environments (e.g. continuous queries)
- Solution*: react fast to maintain performance*

Insufficient memory at runtime

- Memory assigned to some operator(s) is insufficient
  - "The hash table outgrows the memory"
- Causes
  - Memory estimates rely on estimated statistics, whose errors *propagate and amplify* in the physical plan [IC91]
- Consequences
  - OS paging, *very* inefficient
  - Sample performance penalty [BKV98]

% of avail. memory	% degradation
100	0
90	50
80	160
70	350
60	625
50	1200

Levels of adaptation to runtime memory limitations

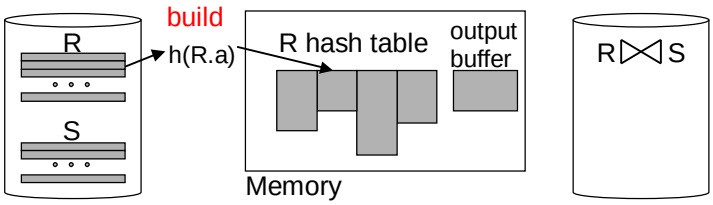
- Physical operators with important memory needs [DeWG85, KNT89, ZG90, PCL93, HN96, IFF+99, BFP+01, MBF+02]
  - Use disk, more efficiently than OS paging
  - *Local adaptation*
- Memory allocation and scheduling [MDeW93, BKV98]
  - Schedule execution to avoid paging
  - *Global adaptation*
- Query re-optimization [KDeW98, IHW01]
  - Change plan to ensure that execution holds in memory
  - *Global adaptation*

Physical operators adapting to insufficient memory

- A success story: the hybrid hash join [DeWG85]
  - Algorithm
  - Applications
- Adaptive hash join algorithms in commercial implementations
- Others (not presented here)
  - Memory-adaptive sorting [PCL93], ...

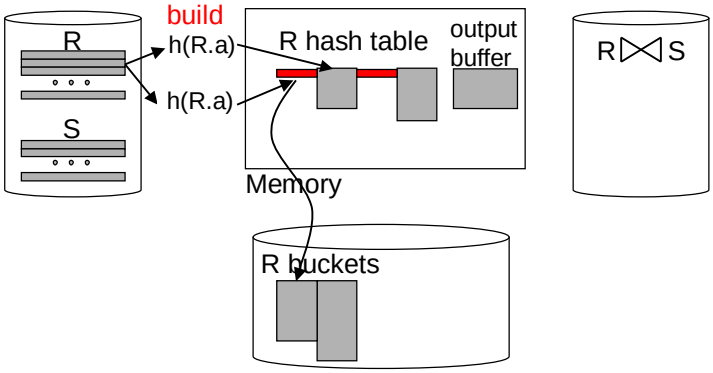
The hybrid hash join algorithm

- Hybrid of regular (optimistic) and Grace (pessimistic) join [NKT86]
- Implemented in: MS SQLServer, IBM DB2, Oracle, ...
- Phase 1: regular hash join



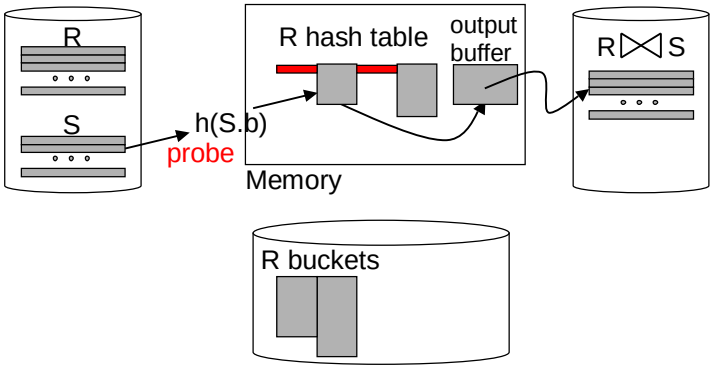
The hybrid hash join algorithm

- Phase 2: hash table reaches memory limit
- Flush some R bucket(s) to disk



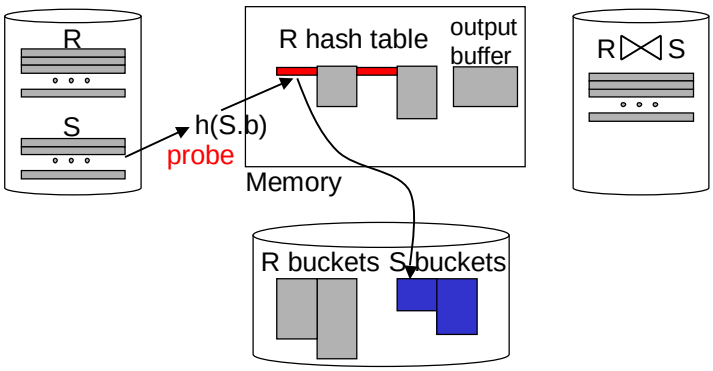
The hybrid hash join algorithm

- Phase 3: R tuples has been exhausted
- Start probing the hash table with S tuples



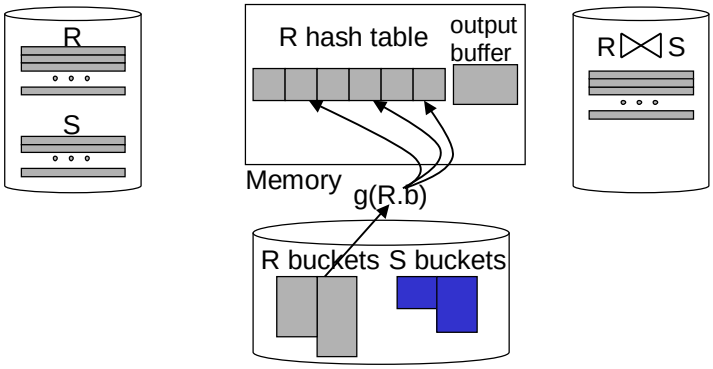
The hybrid hash join algorithm

- Phase 3: R tuples has been exhausted
- Start probing the hash table with S tuples



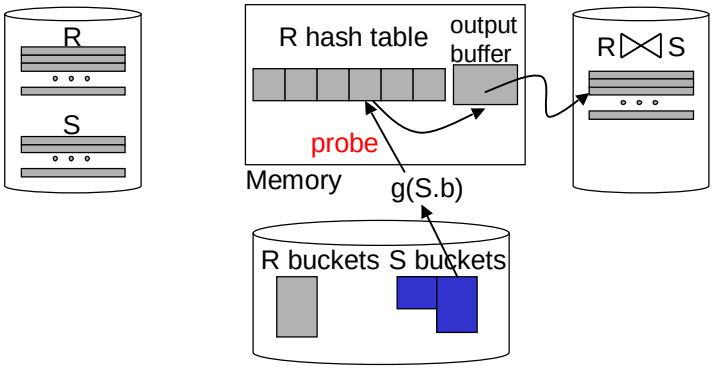
The hybrid hash join algorithm

- Phase 3: S tuples has been exhausted
- Re-hash one disk-resident R partition in memory



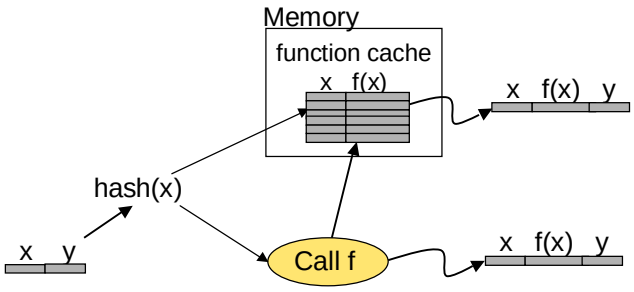
The hybrid hash join algorithm

- Phase 3: S tuples has been exhausted
- Join by pairs R and S overflow buckets



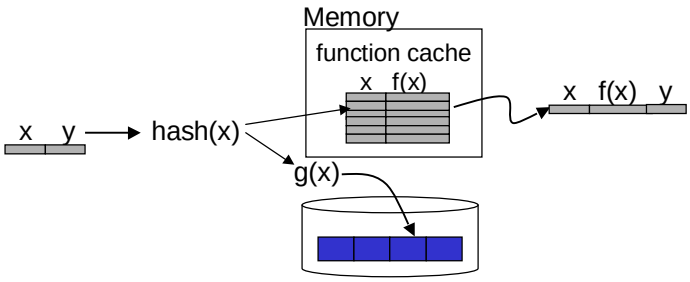
Application of hybrid hash: hybrid cache

- Physical operator for calling user-defined functions using cache [HN96]



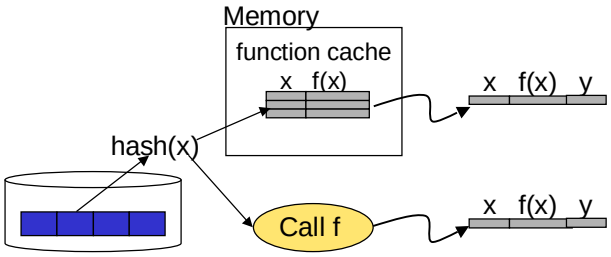
Application of hybrid hash: hybrid cache

- The function cache may outgrow the memory
- Stop calling f; apply second hash function g(x) on (x,y) tuples and write them to disk



Application of hybrid hash: hybrid cache

- When (x,y) tuples are exhausted
  - Discard cache
  - Load one by one buckets from disk, process them building a new cache



Other adaptive variants of hash join

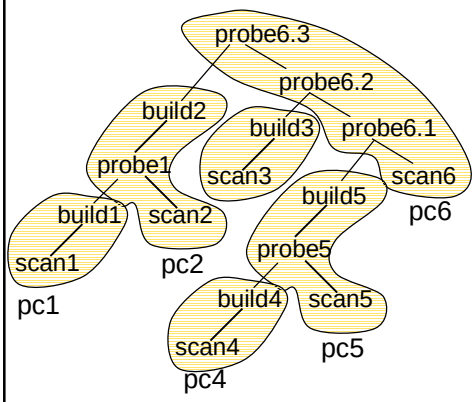
- SQL Server [GBC98]
  - Starts as regular hash join (optimistic)
  - Degrades successively into
    - Hybrid hash join
    - Grace hash join [FKT86] : partition both relations and write all partitions to disk; join partitions by pairs
    - Recursive hash join: if a partition overflows memory, apply a second hash function
  - Also applies: bit vector filtering, role reversal, partition tuning [GBC98]
  - ... Or gets discouraged by too many duplicates and resorts to sort-merge join [Gra00]

Adapting to memory limitations: scheduling and memory allocation

- Adaptive operators *may* run between [M_{min}, M_{max}], run at M_{avail}
- M_{avail} memory assignments to operators may be optimized globally
- Operator scheduling may be optimized globally

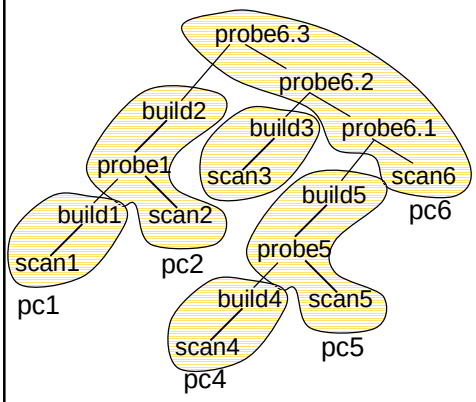
- Adaptive scheduling & allocation [BKP98]:
  - Given a physical QEP and a memory limit
  - Schedule and allocate memory so that
    - Paging is avoided
    - Response time is reduced

Adaptive scheduling and memory allocation



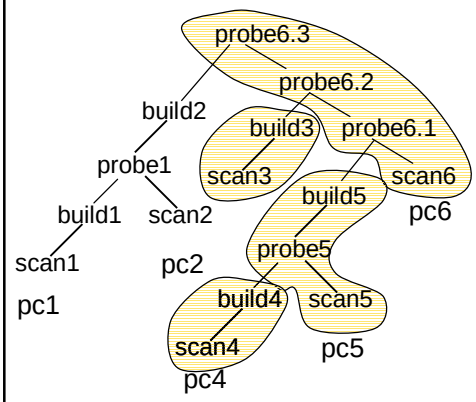
- Choose a pipeline chain scheduling
- Execute in order pipeline chains
- On overflow (building HT_X):
  - If another table HT_Y in memory
    - If HT_Y will be used after HT_X
    - Then write HT_Y to disk
    - Else suspend current pipeline chain and consume HT_Y; then resume
  - Otherwise write HT_X to disk

Adaptive scheduling and memory allocation



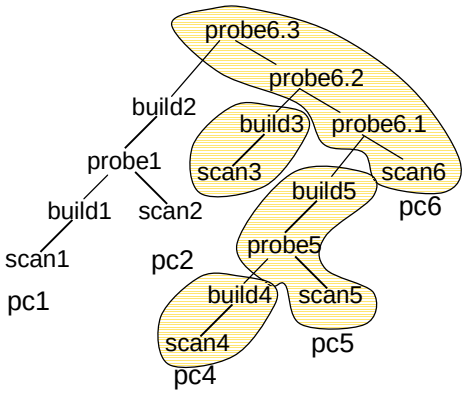
- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc1: HT1 in memory
- Run pc2 using HT1
  - overflow in build2
  - write HT2 to disk, HT1 erased

Adaptive scheduling and memory allocation



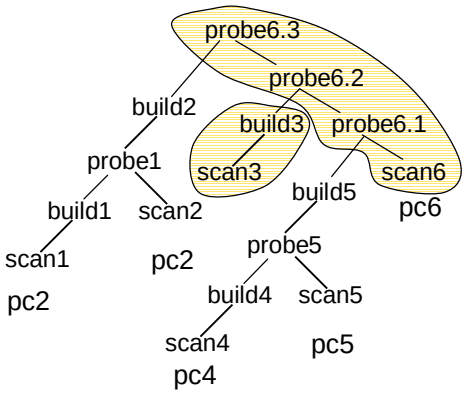
- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc1: HT1 in memory
- Run pc2 using HT1
  - overflow in build2
  - write HT2 to disk, HT1 erased
- Run pc4: HT4 in memory

Adaptive scheduling and memory allocation



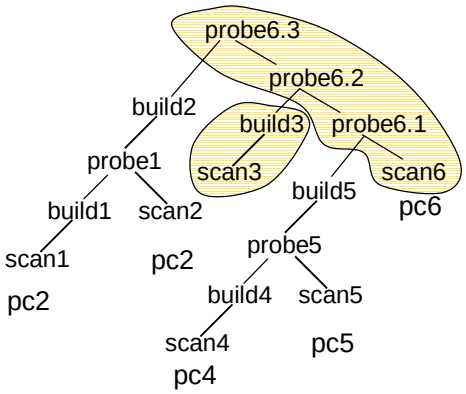
- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc1: HT1 in memory
- Run pc2 using HT1
  - overflow in build2
  - write HT2 to disk, HT1 erased
- Run pc4: HT4 in memory
- Run pc5 using HT4: HT5 in memory, HT4 erased

Adaptive scheduling and memory allocation



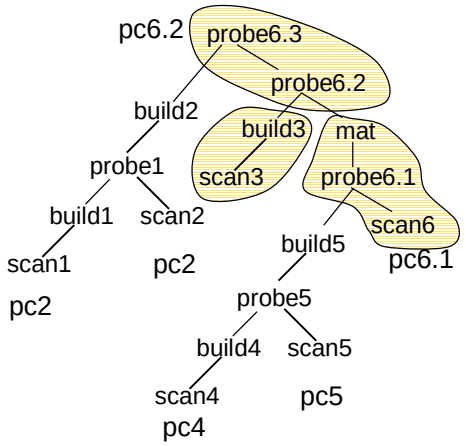
- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc1: HT1 in memory
- Run pc2 using HT1
  - overflow in build2
  - write HT2 to disk, HT1 erased
- Run pc4: HT4 in memory
- Run pc5 using HT4: HT5 in memory, HT4 erased
- Run pc3
  - overflow in build3
  - need to consume HT5

Adaptive scheduling and memory allocation



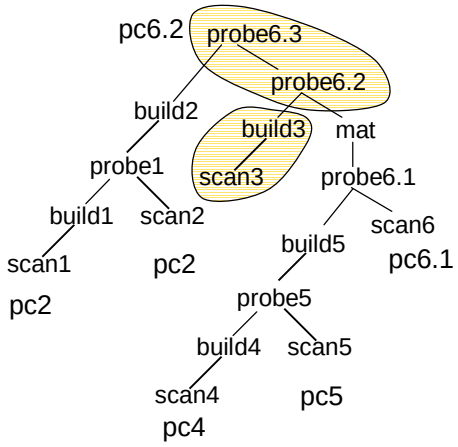
- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc3
  - overflow in build3
  - need to consume HT5
  - suspend pc3
  - cut pc6 in pc6.1 and pc6.2

Adaptive scheduling and memory allocation



- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc3
  - overflow in build3
  - need to consume HT5
  - suspend pc3
  - cut pc6 in pc6.1 and pc6.2
- Run pc6.1 using HT5, HT6.1 on disk, HT5 destroyed

Adaptive scheduling and memory allocation



- [pc1, pc2, pc4, pc5, pc3, pc6]
- Run pc3
  - overflow in build3
  - need to consume HT5
  - suspend pc3
  - cut pc6 in pc6.1 and pc6.2
- Run pc6.1 using HT5, HT6.1 on disk, HT5 destroyed
- Resume pc3, HT3 in memory
- Run pc6.2 using HT6.1 and HT3

Adapting to memory limitations: summary

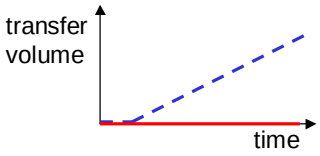
- React *during* query execution
- Adaptive operators
  - Relatively easy to implement, local adaptation
  - Adopted in industrial products
  - *"Degrades gracefully with limited memory"* generally required
- Adaptive scheduling and memory allocation
  - More complex, better global control
  - Scheduling requires suspending and resuming pipeline chains

Unexpected data transfer rate (1/2)

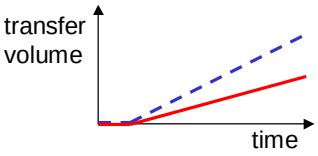
- In distributed settings
  - Distributed DBMSs, wrapper-mediator systems
- Causes:
  - Remote site or network failure
  - Variations in network bandwidth
  - *Any variation on a remote site* (load, memory,...)
- Consequences:
  - Impossibility to answer query (failures)
  - Idle times: operator waits for input
    - Increased query response time

Unexpected data transfer rate (2/2)

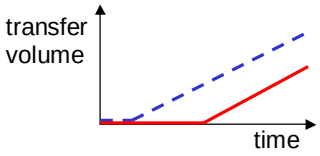
- Data trasfer rate (——) is different from expectation (- - -)
  - Blocked (site or link failure)



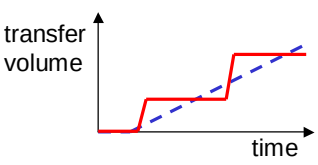
  - Slow



  - Delayed



  - Bursty

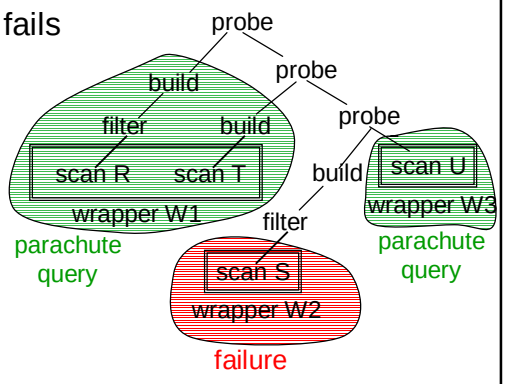


Adapting to unexpected transfer rates

- Adapting to failure
  - Evaluate the feasible part of the query, evaluate the query later using the partial results (*parachute queries* [BT98])
- Adapting to delays
  - While a source is blocked, schedule some other work, perhaps re-optimize (*query scrambling* [AFT+96, UFA98])
- Adapting to delays, slow or bursty transfer
  - Dynamic scheduling [BFM+00]
  - Adaptive operators: double pipelined join [WA91, IFF99], XJoin [UF00]; also Eddies [AH00, MSH+02]

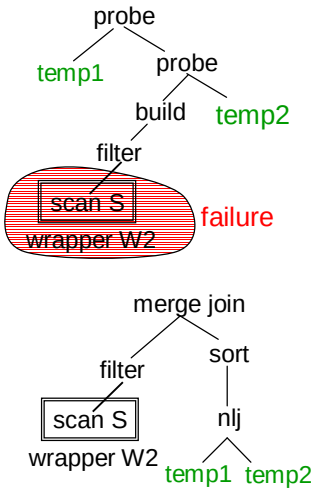
Parachute queries

- Context: wrapper-mediator system
- Wrapper on remote site fails
  - Parachute queries: executable fragments of the plan
  - Materialize results as temporary relations



Parachute queries

- When remote source becomes available, run *incremental* query on
  - Results of parachute queries
  - Remote source
- Incremental query is re-optimized
  - Needs query rewriting using views
- Sensitive to timeout for failure detection



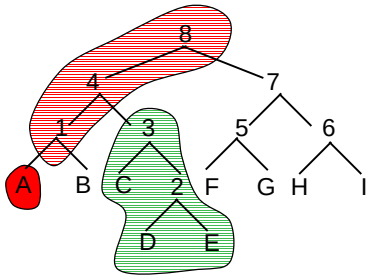
Query scrambling

- Changes scheduling to hide delayed sources
  - Blocked for a while, then available
  - A delayed source blocks a set of operators in the QEP
  - *Run some other non-blocked operators while waiting for the delayed source*
- *Runnable subtree*
  - QEP subtree whose operators do not depend on delayed sources or blocked operators
- Two phases:
  - Re-scheduling
  - Re-optimization

Query scrambling in presence of delayed sources

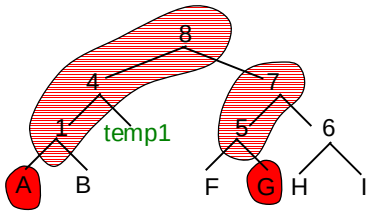
- Re-schedule:
  - Run the next scheduled runnable subtree, materialize the result
  - After processing a runnable subtree
    - If delayed data started to arrive, revert to normal
    - Otherwise, pick another runnable subtree
    - When no runnable subtrees are left, re-optimize
- Re-optimize: combine materialized results via new operators
  - After executing an operator
    - If delayed data started to arrive, revert to normal
    - Otherwise, re-optimize

Query scrambling in the presence of delayed sources

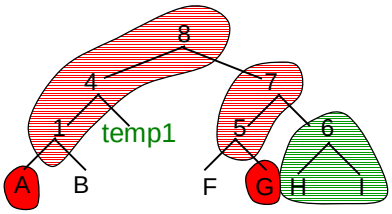


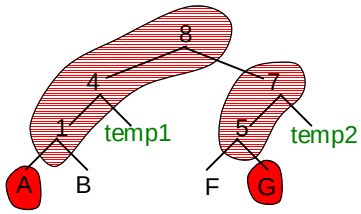
- Starts from the scheduling dictated by iterators [1,2,3,4,5,6,7,8]
- A delayed: 1, 4, 8 blocked
- Identify next runnable subtree in the scheduling, materialize it

Query scrambling

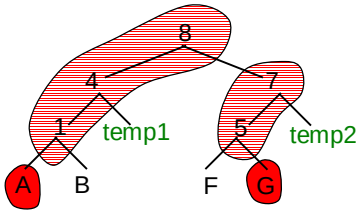


- In the meantime, G becomes unavailable, 5 and 7 blocked
- Identify next maximum runnable subtree, materialize it
- Nothing left to run: re-optimize

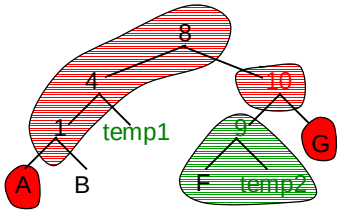


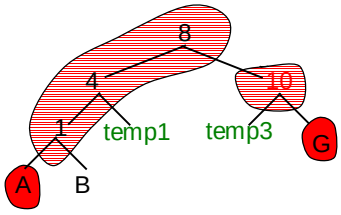


Query scrambling

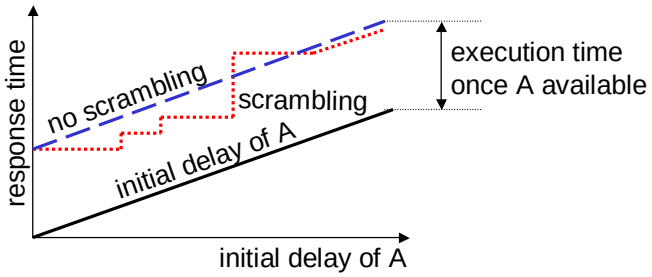


- Re-optimization: join F and temp2
- Nothing left to do: block, waiting for data





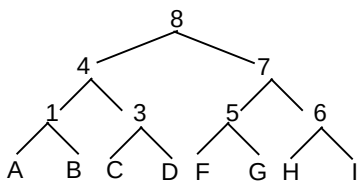
Effect of query scrambling



- Optimization has a high overhead
  - decision to scramble *bets on the future*
- Very sensitive to timeout value

Effect of query scrambling

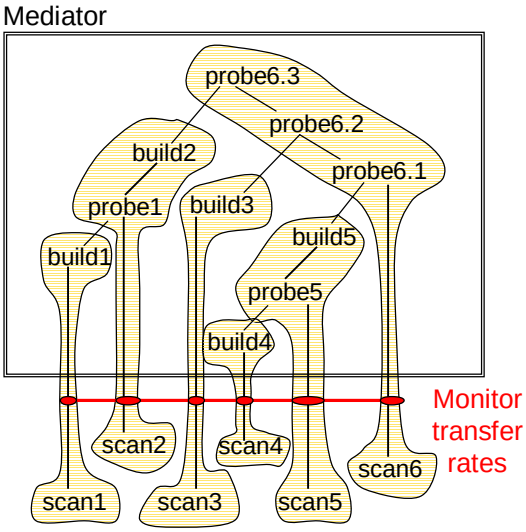
- Strongly influenced by the iterator-dictated scheduling
  - If H is the first source delayed, nothing left to scramble



Dynamic scheduling

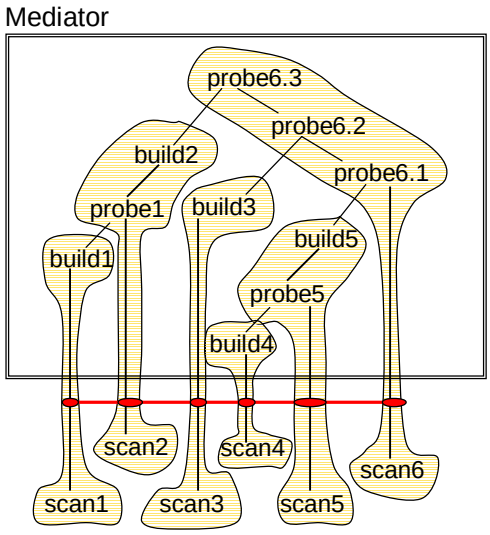
- Attempts to find an optimal scheduling with respect to
  - Delays
  - Bursty arrival
  - Slow arrival
- The network is the bottleneck
- *Interleaves* execution of many concurrent pipeline chains, limited by
  - Producer-consumer dependencies
  - Available memory
- Give priority to *critical pipeline chains*: those processing data faster than it arrives

Dynamic scheduling



- Pipeline chains are ordered according to their critical degree
  - The order is recomputed if transfer rates vary significantly
- Scheduling:
  - Process one *batch of tuples at a time* from the *most critical pipeline chain (mcp)*

Dynamic scheduling



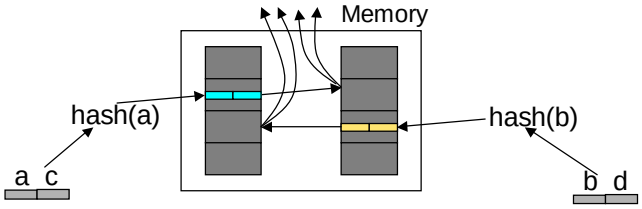
- Scheduling (cont'd):
  - If mcp does not hold in memory, cut it as high as possible, run lower fragment, materialize
  - If mcp cannot run because of dependencies, cut it, materialize source data
  - Interleaves the execution of concurrent pipeline chains
- More general than scrambling
  - More complex
- Lower overhead

Operators adapting to unexpected transfer rates

- Context
  - Remote (hash) join processing
- Goal
  - Transfer rates for build and/or probe inputs may vary
  - Avoid stalling
- Solutions
  - Double pipelined hash join
  - XJoin

The double pipelined hash join

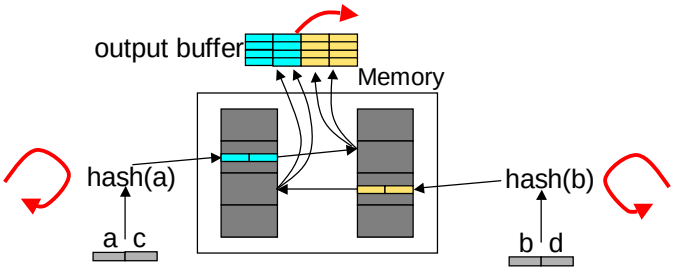
- The goal: avoid stalling while the build input is slow
  - The transfer rates of build and probe inputs may vary
  - Build *both relations* at the same time



  - On arrival, each tuple is built and probes
    - Non-blocking on both sides

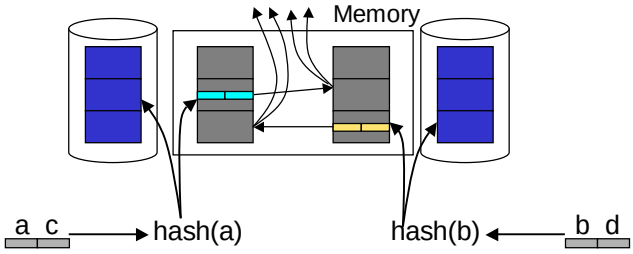
The double pipelined hash join

- Blocks only when both inputs are blocked
- Bigger memory needs (*two* hash tables)
  - Adapts gracefully to memory limitations [IFF+99]
- Needs 3 threads to conform to the iterator model



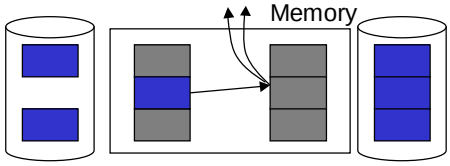
The XJoin

- May work even with both inputs blocked
- Needs less memory: each bucket resides partially on disk



The XJoin

- When both inputs are blocked
  - bring in memory one disk-resident bucket part
  - probe a memory-resident bucket part



- One disk-resident bucket part may be brougt in memory many times
  - Tuple timestamps to ensure correctness

Unexpected data characteristics at runtime (1/2)

- Occurence: source data or intermediate results
- Causes
  - Existing statistics are very imprecise
    - Commercial systems: significant research on histograms
    - Impossible to construct *all* histograms
    - Continued use of "magic numbers"
      "$R \bowtie_{R.a=S.b} S$ returns $N_R * N_S * 0.1$ tuples"
    - Wrapper-mediator systems: data statistics most difficult to obtain
  - Source data has changed since the last statistics gathering

Unexpected data characteristics at runtime (2/2)

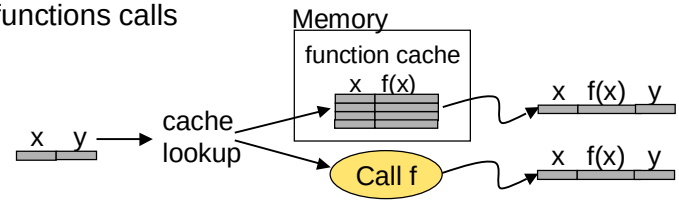
- Occurence: source data or intermediate results
- Consequences
  - Operators' data structures may not hold in the memory that was assumed available for them
    - The choice of the physical plan is wrong
    - *Memory-adaptive solutions apply*
  - Data is transmitted in bursts between operators
    - Idle then busy periods ("variable selectivity")
    - Increased response time

Unexpected data characteristics: adaptive solutions

- Adaptive operators [BFP+01, MBF+02]
- Change the physical query plan
  - Build a limited degree of choice in physical query plans and choose at runtime [GW89, GC94]
  - Gather statistics during execution and re-optimize if needed [KDeW98, IFF+99, IHW01]
- Give up the physical query plan
  - Allow different processing orders for *each tuple* [AH00, MSH+02]

Operators adapting to data statistics

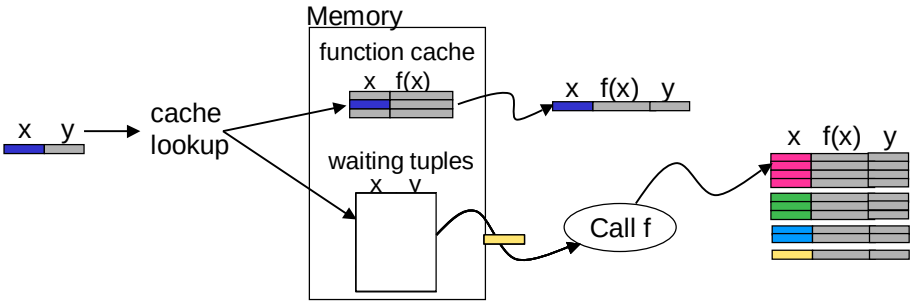
- Early Rate BindJoin [MBF+02]: operator for expensive functions calls



- Data output rate tends to be:
  - Slow at the beginning (cache empty, all values have to be processed): small *early rate*
  - Fast towards the end (results are available in cache)
- Large early rate is desirable

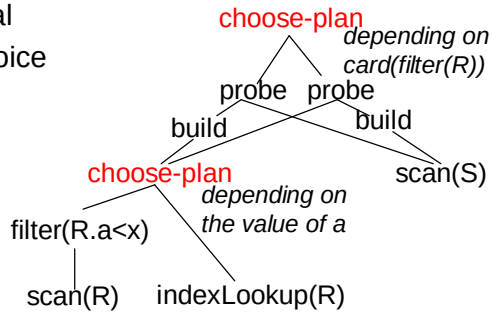
Early rate BindJoin

- Solution:
  - Accumulate arguments in internal buffer
  - Call function on most frequent values first



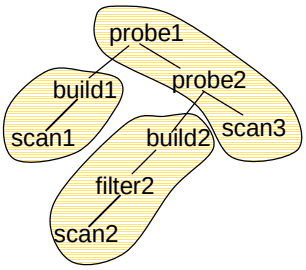
Dynamic query execution plans

- Goal: use 1 query execution plan for several similar user queries (avoid re-optimising) [GW89, GC94]
- For queries containing user-supplied constants, different plans may be optimal
  - Allow runtime choice
- Adaptive within the set of specified options



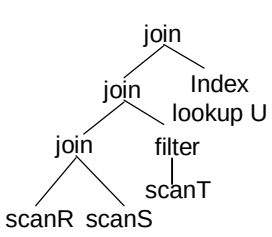
Mid-query re-optimization

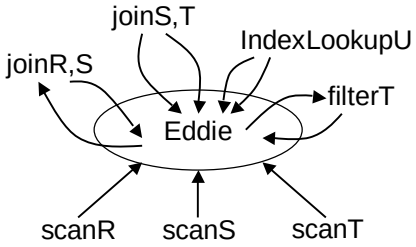
- Gather statistics during execution, use them to optimize the remaining work [KDeW98]
- While executing a pipeline, collect
  - Cardinality, size, min and max for *every* intermediate result
  - Statistics with a high *innacuracy potential*
    - Current estimate suspected wrong
- At the end of the pipeline
  - Re-estimate cost based on new statistics
  - If very bad, re-optimize
- [IHW01] takes similar approach, re-optimizes *within* pipeline



Eddies: per-tuple operator reordering

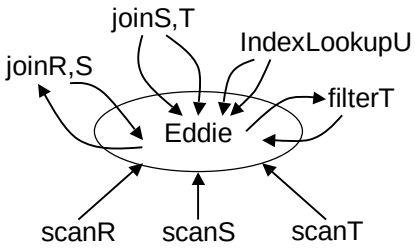
- Context: wrapper-mediator system [AH99]
  - Unknown or variable operator selectivities
  - Variable tuple transfer rates
- Solution: replace the query plan with an Eddie
  - Routes each tuple on a potentially different path





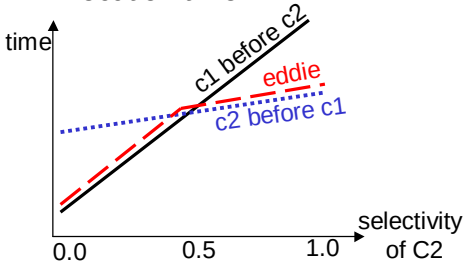
Eddies: per-tuple operator reordering

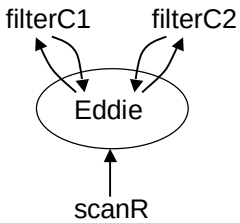
- Join inputs may switch correctly only at *moments of symmetry*
  - Standard hash join: never
  - Double pipeline join: at any point (instance of Ripple Join [HH99])
- Uses bitmaps to keep track of completion of each tuple
- Routing policy to give tuples to competing operators
  - Favors operators who *drain* tuples, i.e., fast and selective



Eddies: benefits of adaptativity

- Experiment: select * from R where c1(R.a) and c2(R.b)
- Selectivity of c1: 0.5; selectivity of c2: varies from 0 to 1
- Execution time:



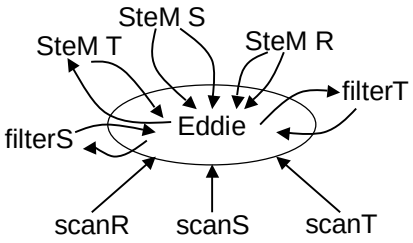


- Finds best execution order *without a static plan*

Eddies and SteMs

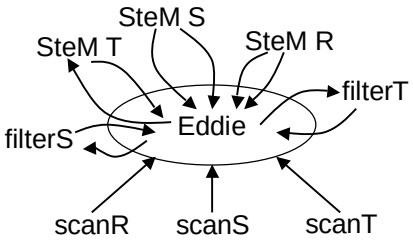
- Context: long-running queries over streams [MSH+02]
- *No estimate stays correct during query lifetime*
  - Drop query plans alltogether
  - Use 1 Eddie + 1 State Module per source

filter(S, filter(T, join(R,S,T)))



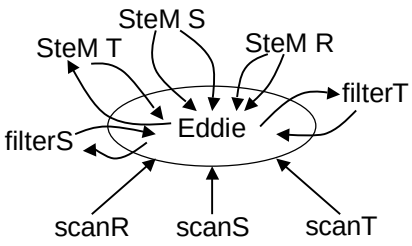
Eddies and SteMs

- Eddie + State Modules: multi-way double pipelined join
- Each tuple
  - Is built into the corresponding SteM
  - Probes in any order other SteMs
- Advantage: factorization
  - For all queries on R, one SteMR, one filterR



Eddies and SteMs: ensuring correctness

- Many, many more bitmaps
- Each tuple must be *built before it probes*
- Two tuples may (still) erroneously join an unbound number of times.
  - Timestamp every tuple
  - Joined tuple is correct iff *build component is older than probe component*
    - *"build, then probe"*
  - Eddie kills incorrect tuples
    - Unknown overhead



Adapting to unexpected data characteristics: summary

- The problem appears in *all* query processing scenarios
  - Most difficult for wrapper-mediator systems
  - In stream processing, statistics are ill-defined
- Granularity of adaptive solutions
  - Dynamic query optimization / scheduling
  - Operator level
  - Per-tuple adaptivity: local, centralized approach
- Commercial systems attempt to refine their statistics
  - Over longer time intervals, off-line (to be seen)

Adapting to unexpected events at runtime: summary (1/2)

- Unexpected events at runtime:
  - Insufficient memory
  - Data transfer rates
  - Data characteristics
- Adaptive mechanisms incorporated in
  - Regular operators (e.g. Hybrid Hash Join, DPHJ, XJoin)
  - Special operators (e.g. “choose plan”, Eddy)
  - Scheduler (e.g. query scrambling, dynamic query scheduling)
  - Runtime control: gather statistics, re-invoke the optimizer (mid-query re-optimization)

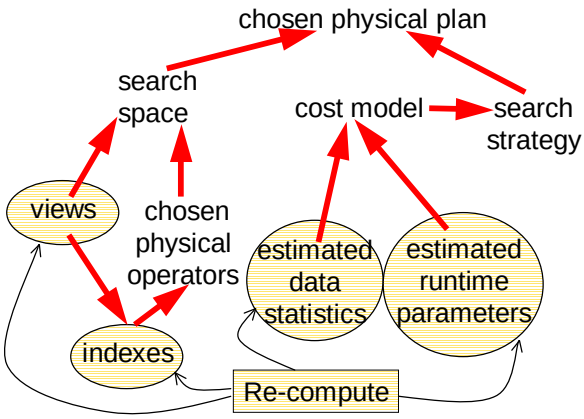
Adaptivity at runtime: summary (2/2)

- Memory-adaptive operators: success in industrial systems
- Delay-adaptive operators: useful in wrapper-mediator systems
- Some thoughts of Goetz Graefe
  - at U.Portland, U.Oregon: dynamic query evaluation plans [GW89, GW94]
  - at Microsoft [Gra00]
    “In modern systems [..] there are many adaptive techniques [...] typically ignored in the cost functions of commercial query optimizers, partially because they are too difficult to incorporate, and partially because a sufficient strong case for incorporating them has not been made. What does that say about techniques as adaptive as dynamic query evaluation plans ?”

Long-term adaptativity: learning about the unknown

- Optimizer knowledge is wrong or incomplete, but *stable, correct values exist*
  - Data statistics
  - Set of useful statistics, indexes
  - Data transfer rates
- Typical in centralized or distributed DBMS
- *Refine optimizer knowledge to improve performance*

Long-term adaptativity: learning the unknown



Learning about data characteristics

- Indexes and statistics are:
  - Chosen for a given workload
    - Typical DBA task, part of *database tuning* [Sha]
    - Recent DBMSs (DB2, SQL Server) *recommend* or *choose* them [AA]
  - Built
    - From scratch, after significant data changes [SAC+79]
    - Maintained by gathering information while running queries [SLM+01]

The AutoAdmin project

- Purpose: make DBMS (MS SQL Server) self-tuning to reduce cost of ownership
- Given a workload of queries $\{Q_1, \ldots Q_n\}$ and a DBMS, automatically choose:
  - Indexes [CN97], statistics [CN00], materialized views and indexes [ACN00], statistics on intermediate results [BC02]
- Minimizing the *estimated cost* of the workload:
  $\Sigma_i \text{OptimizerEvalCost}(Q_i)$
- *Indexes etc. are good only if the optimizer uses them*

AutoAdmin: outline of the index selection procedure (1/3)

- Given a workload W={Q_1, Q_2, ..., Q_n}
- Choose a *configuration* (set of indexes) of size k minimizing the *estimated cost* of the workload
- Search space potentially huge
  - Avoids asking the optimizer to evaluate all possible configurations [CN97]
  - Usage of heuristics

AutoAdmin: outline of the index selection procedure (2/3)

1. Choose C, a set of *one-column candidate indexes* for W:
   a) Choose one-column candidate indexes for every Q_i
   b) Candidate indexes for W: $\cup_i$(candIndexes(Q_i))
   Heuristic: an optimal index for the workload has to be an optimal index for at least one Q_i
2. Choose C_k = best k one-column indexes from C; let C = C_k.

Up to now, C only contains one-column indexes !!!

AutoAdmin: outline of the index selection procedure (3/3)

How to choose indexes on several columns:

3. For idxSize=2,...,maxIndexSize
   a) Let newCand = { idx(col_1, col_2, ..., $col_{idxSize}$) such that idx(col_1, col_2, ..., $col_{idxSize-1}$) $\in$ C }
   b) Add newCand to C
   c) Choose C_k = best k one-column indexes from C
      Heuristic: a good index on idxSize columns is an "extension" of a good index on idxSize-1 columns
      - The prefix of a good index is a good index

Details: choosing one-column candidate indexes for query Q_i

- Only on attributes used in the query
  - select * from R, S where R.a=S.b and R.c between 0 and 7: consider only R.a, S.b, R.c
  - *Heuristic: query engines do not use more than*
    - *j indexes for a single table, j=1 or 2*
    - *indexes on more than t tables for a given query, t=2*

Details: choosing one-column indexes for the workload

- *Candidate indexes* for W: $C=\cup_i(\text{candIndexes}(Q_i))$, size(C)=n

  If n is larger than the limit k, need to prune

- Choose best k indexes from C: many possible configurations
- *Heuristic search:*
  - *Explore all configurations of size m, m<k (m=2)*
  - *Let C_m be the best configuration of size m*
  - *Apply a greedy algorithm to add the most profitable k-m indexes from C*

AutoAdmin: choosing materialized views *and* indexes for a workload

- Given a workload W={Q_1, Q_2, ..., Q_n} choose a *configuration* of
  - Indexes
  - Materialized views
  - Indexes on materialized views
- ...occupying less than S space
- Minimizing the estimated cost of the workload

AutoAdmin: why choose materialized views and indexes together ?

- They are redundant structures that speed up query processing
  - The presence of an index may change the utility of a materialized view
  - Proposing indexes and views separately may lead to redundancy
  - Views should be selected first... blocking the proposal of interesting indexes
- They compete for the same resource: space
  - Allocating φ*S for indexes and (1-φ)*S for views is suboptimal

AutoAdmin: choosing materialized views and indexes (1/2)

1. Choose a set of candidate views
   a) Identify sets of *interesting table subsets* $T=\{T_1, ..., T_r\}$
      - Materializing views on T significantly reduces the cost of the workload
   b) For each interesting table subset propose
      - A view cummulating *all joins and selections on T appearing in* Q_i
      - (If some Q_i performs aggregation) a similar view with *aggregation*
   c) Merge similar views into more general ones

AutoAdmin: choosing materialized views and indexes (2/2)

2. Choose a set of candidate indexes (seen)
   - on tables *and* materialized views
3. From the *n* candidate indexes and materialized views, greedily select the most profitable ones until the space limit S is reached

AutoAdmin: choosing statistics for a workload

- Given
  - a query Q
  - The set S_0 of *syntactically relevant statistics*
    - On all join and selection columns (too large)
- Choose a set C of at most k statistics such that
  - The cost estimates of the QEPs chosen by the optimizer for (W, S_0) and (W, C) are close
    - Typical value: within 20% range

AutoAdmin: choosing statistics for a workload

- Start with no statistics (C empty)
- While *(more statistics are needed)*
  - Identify *the most important statistic to build*
  - Add it to C

AutoAdmin: when are statistics needed ?

- When statistics are missing, the optimizer uses *magic numbers* (selectivity variables s_1, s_2, ... s_n.)
- The optimizer's estimate for the cost of a query Q is monotonic in the values of s_1, s_2, ..., s_n.
- Let S_x be a set of statistics and $\varepsilon \approx 0$.
  - P_{low}: the optimizer's chosen QEP for Q, using S_x, if $s_1=s_2=...=s_n= \varepsilon$
  - P_{high} : the same for $s_1=s_2=...=s_n= 1 - \varepsilon$
- If P_{low} and P_{high} are close enough, S_x contains enough statistics

AutoAdmin: which statistics are most important ?

- For a given query Q:
  - Find *most expensive operator op* in the QEP proposed by the optimizer for Q
    - Maximizing cost(op) - Σ(cost children of op)
  - Consider statistics for *op*

Learning data characteristics: summary

- Going towards self-tuning DBMSs
  - The DBMS adapts to the workload
- Complex algorithms implemented in commercial products
  - Heavy use of heuristics and rules of thumb
- As indexes, statistics, and materialized views get smarter, optimizer's estimates get better
  - *Long-term and short-term adaptativity are competing*
  - (In centralized industrial systems) long-term is a more robust choice

Learning transfer times

- Context: wrapper-mediator systems
- Network transfer times vary a lot, depending on:
  - Day of the week
  - Time of day
  - Quantity of data transferred
- WebPT [RZB+99]: Web Prediction Tool
  - Monitor transfer rates while executing queries
  - Refine knowledge about transfer rates based on experience

WebPT: learning network transfer times

- Gather query feedback in *cells* along the dimensions *D*ate, *T*ime, *Q*uantity.
- Start with a single cell [Monday-Sunday], containing a static estimate of transfer rates
- Every query execution yields a query feedback [D, T, Q, rate]
  - If rate is different from the estimate of the cell containing [D, T, Q]
    - Split the cell in two
    - Adjust the estimates of the new cells
  - Otherwise, increase confidence of the cell

WebPT: example

Day	Monday-Friday						Saturday-Sunday
Time	8am-2pm		2pm-8pm			8pm-8am	8am-8am
Qty	<100K	>100K	<100K	<700K	>700K	any	any
	v1	v2	v3	v4	v5	v6	v7

- Query feedback at 12am on Saturday, different from cell estimate:

Day	Monday-Friday	Saturday		Sunday
Time	(the same)	8am-12pm	12pm-8am	12am-8am
Qty		any	any	any
	v1 ... v6	v8	v9	v10

Learning the unknown: summary

- Robust methods exist for learning
  - Values of data statistics
  - For a given workload, the optimal sets of
    - Statistics
    - Materialized views
    - Indexes
  - Data transfer times
- Off-line learning has less overhead than run-time reacting, but similar goals

Adaptive query processing: summary

- Heterogeneous mix of technologies
- Comparisons possible among common dimensions
  - Double pipelined join vs XJoin
- *No common testbed to compare relative and combined efficiency*
  - If statistics are known, how useful is memory adaptiveness ?
  - If transfer rates are known, how useful is query scrambling ?
- From innovative, extremely new techniques to strong, proven industrial implementations

Remember the goal: performance

- Thoughts of Goetz Graefe [Gra00]:

  "An improvement measured by a small factor, say 3, is laudable and useful, but not a breakthrough - improvement in hardware technology will give us the same [...] in just one or two years.
  In order to be truly a breakthrough, a performance improvement has to be measured in orders or magnitude. Materialized views are one such technique. Dynamic query plans, on the other hand, so far have not achieved this level of success on a broad scale.
  Can we achieve consistent and predictable order-of-magnitude improvements for database systems by combining dynamic query plans with on-the-fly indexing and materialized views ?"

References

- [AA] The AutoAdmin Project. http://research.microsoft.com/dmx/autoadmin
- [ABC01] V.Aguilera, S.Boiscuvier, S.Cluet. “Pattern Tree Queries in Xyleme”. INRIA Technical Report, 2001.
- [ACN00] S.Agrawal, S.Chaudhuri, V.Narasayya. “Automated Selection of Materialized Views and Indexes for SQL Databases”, VLDB 2000.
- [AFT+96] L.Amsaleg, M.Franklin, A.Tomasic, T.Urhan. “Scrambling Query Plans to Cope with Unexpected Delays”, PDIS 1996.
- [AH00] R. Avnur, J.Hellerstein. “Eddies: Continuously Adaptive Query Processing”, SIGMOD 2000.
- [BC02] N.Bruno, S.Chaudhuri. “Exploiting Statistics on Query Expressions for Optimization”. SIGMOD 2002.
- [BFP+01] L.Bouganim, F.Fabret, F.Porto, P.Valduriez. “Processing Queries with Expensive Functions and Large Objects in Distributed Mediator Systems. ICDE 2001.

References

- [BFM+00] L.Bouganim, F.Fabret, C.Mohan, P.Valduriez. “Dynamic Query Scheduling in Data Integration Systems”. ICDE 2000.
- [BKV98] L.Bouganim, O.Kapitskaia, P.Valduriez. “Memory-Adaptive Scheduling for Large Query Execution”, CIKM 1998.
- [BT98] P.Bonnet, A.Tomasic. “Parachute queries in the presence of unavailable data sources”. Technical Report RR-3429, INRIA, 1998.
- [CN97] S.Chaudhuri, V.Narasayya. “An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server”. VLDB 1997.
- [CN00] S.Chaudhuri, V.Narasayya. “Automating Statistics Management for Query Optimizers”, ICDE 2000.
- [DeWG85] D.DeWitt, R.Gerber. “Multi-processor Hash-based Join Algorithms”, VLDB 1985.
- [FKT86] S. Fushimi, M. Kitsuregawa, H.Tanaka. “An Overview of the System Software of a Parallel Relational Database Machine GRACE”. VLDB 1986.

References

- [GBC98] G.Graefe, R.Bunker, S.Cooper. “Hash Joins and Hash Teams in Microsoft SQL Server”, VLDB 1998.
- [Gra90] G.Graefe. “Encapsulation of Parallelism in the Volcano Query Processing System”, SIGMOD 1990.
- [Gra93] G.Graefe. “Query Evaluation Techniques for Large Databases”, ACM Computing Surveys 25(2), 1993.
- [Gra00] G.Graefe. “Dynamic Query Evaluation Plans: Some Course Corrections ?”, IEEE Data Engineering Bulletin, 2000.
- [HFC+00] J.Hellerstein, M.Franklin, S.Chandrasekaran et al. “Adaptive Query Processing: Technology in Evolution”, IEEE Data Engineering Bulletin, 2000.
- [HH99] P.Haas, J.Hellerstein. “Ripple Joins for Online Aggregation”. SIGMOD 1999.
- [HN96] J.Hellerstein, J.Naughton. “Query Execution Techniques for Caching Expensive Methods”, SIGMOD 1996.

References

- [IC91] I.Ioannidis, S.Christodoulakis. “On the Propagation of Errors in the Size of Join Results”, SIGMOD 1991.
- [IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, D.Weld. “An Adaptive Query Execution System for Data Integration”, SIGMOD 1999.
- [IHW01] Z.Ives, A.Halevy, D.Weld. “Convergent Query Processing”, submitted for publication, 2001.
- [KNT89] M.Kitsuregawa, M.Nakayama, M.Takagi. “The Effect of Bucket Size Tuning in the Dynamic Hybrid Grace Hash Join Method”. VLDB 1989.
- [MBF+02] I. Manolescu, L.Bouganim, F.Fabret, E.Simon. “Efficient Data and Program Integration Using Binding Patterns”. BDA 2002.
- [MDeW93] M.Mehta, D.DeWitt. “Dynamic Memory Allocation for Multiple-Query Workloads”, VLDB 1993.
- [MSH+02] S. Madden, M. Shah, J. Hellerstein, V.Raman. “Continously Adaptive Continuous Queries over Streams”. SIGMOD 2002.
- [PCL93] H.Pang, M.Carey, M.Livny. “Memory-adaptive External Sorting”. VLDB 1993.

References

- [RZB+99] L.Raschid, V.Zadorozhny, L.Bright, T.Zhan. “A Comparison of a Web Prediction Tool and a Neural Network in Learning Response Time for WebSources using Query Feedback”. CoopIS 1999.
- [SAC+79] P.Sellinger, M.Astrahan, D.Chamberlin, R.Lorie, T.Price. “Access Path Selection in a Relational Data Management System”, SIGMOD 1979.
- [Sha] Dennis Shasha. “Database Tuning”, 2nd edition, 2002.
- [SLM+01] M.Stillger, G.Lohman, V.Markl, M.Kandil. “LEO - DB2’s Learning Optimizer”, VLDB 2001.
- [UFA98] T.Urhan, M.Franklin, L.Amsaleg. “Cost-Based Query Scrambling for Initial Delays”, SIGMOD 1998.
- [WA91] A.Wilschut, P.Apers. “Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991.
- [YC93] P.Yu, D.Cornell. “Buffer Management Based on Return on Consumption in a Multi-Query Environment”. VLDB Journal, 1993.
- [ZG90] H.Zeller, J.Gray. “An Adaptive Hash Join Algorithm for Multiuser Environments”, VLDB 1990.