Algebraic Operators

Algebraic Operators

Queries are translated into relational algebra

- therefore a DBMS must offer implementations for all algebraic operators
- often more than one implementations
- different implementations are tuned for different usage scenarios

Complexity varies

- a few operators are very simple
- but most are quite complex
- pipelining operators tend to be simple
- pipeline breakers tend to be complex

Table Scan

The most basic operator

- produces all tuples contained in a relation
- conceptually very simple
- implementation complexity varies from simple to complex
- has to navigate the physical representation of the relation
- additional complexity from deferred updates, snapshot isolation, etc.

```
TableScan::produce()
  for each page extent e
    for each page p in e
      fix p
      for each tuple t in p
        consumer.consume(t)
      unfix p
```

Selection

A selection σ_p

- filters out all tuples that do not satisfy p
- a very simple operator
- many systems do not even implement it as separate operator
- instead, piggybacked onto other operators

Select::produce()
  input.produce()

Select::consumer(t)
  if $p(t)$
    consumer.consume(t)

Map

A map $\chi_{a:f}$

- computes a new column by evaluating f
- another very simple operator
- like selections, often piggybacked

Map::produce()
  input.produce()

Map::consumer(t)
  $t' = t \circ [a : f(t)]$
  consumer.consume(t')

Join

A join $e_1 \bowtie_p e_2$

- a very complex operator
- one of the most important operators
- several different implementations exist

Candidate implementations depend on the join itself:

1. if e_2 depends upon e_1, nested loop join must be used (i.e. dependent join $e_1 \bowtie e_2$)
2. otherwise, if p has the form $e_1.a = e_2.b$, any join algorithm can be used (equi-join)
3. otherwise, either nest loop or blockwise nested loop can be used

Nested-Loop Join

The nested loop join $\bowtie_p^{NL}$ is the most flexible, but also most simple and inefficient join

- evaluates the right hand side for every tuple of the left side
- pairwise comparison, suitable for any kind of predicate
- the right hand side is evaluated very frequently

NLJoin::produce()
  left.produce()

NLJoin::consumeFromLeft(t)
  $t_L = t$
  right.produce()

NLJoin::consumeFromRight(t)
  $t' = t_L \circ t$
  if $p(t')$
    consumer.consume(t')

Blockwise-Nested-Loop Join

The blockwise nested loop join $\bowtie_p^{BNL}$

- loads as many tuples from the left side as possible
- evaluates the right side and joins
- and repeats this with additional chunks from the left side
- like a NL join, suitable for any predicate (but not for $\rtimes$)
- greatly reduces the number of passes over the right hand side
- potentially speeds up execution by orders of magnitude

BNLJoin::produce()
  clear the tuple buffer
  left.produce()
  if tuple buffer is not empty
    right.produce()

Blockwise-Nested-Loop Join (2)

BNLJoin::consumeFromLeft(t)
  if not can materialize t
    right.produce()
    clear the tuple buffer
  materialize t in tuple buffer

BNLJoin::consumeFromRight(t)
  for each t_L in the tuple buffer
    $t' = t_L \circ t$
    if $p(t')$
      consumer.consumer(t')

Sort-Merge Join

The sort-merge join $e_1 \bowtie_p^{SM} e_2$

- assumes that p has the form $e_1.a = e_2.b$, that e_1 is sorted on a, and that e_2 is sorted on b
- some implementations actually perform the sort, too, but we consider it as separate operator
- performs a linear pass over the input to find matching entries
- very fast and efficient (after sorting)
- **but:** only simple for the $1 : N$ case!

Note: a SM join is simple in the iterator model, but not in the push model (control flow). We materialize the left hand side here to simplify the code, which is usually not needed.

Sort-Merge Join (2)

SMJoin::produce()
  prepare a temp segment for spooling
  left.produce()
  l_L =first spooled tuple
  right.produce()

SMJoin::consumeFromLeft(t)
  spool t to temp segment

Sort-Merge Join (3)

SMJoin::consumeFromRight(t)
while $(*I_L).a < t.b$
advance I_L
$I_B = I_L$
while $(*I_B).a = t.b$
consumer.consume($(*I_B) \circ t$)
advance I_B

- non-standard, as the left hand side is already materialized
- one usually tries to avoid this
- better: parallel scans through both sides
- materialization only if an $n : m$ match is found

Hash-Join

The hash join $e_1 \bowtie_p^{HJ} e_2$

- assumes that p has the form $e_1.a = e_2.b$
- builds a hash table from e_1, and probes the hash table with e_2
- in-memory case and external memory case
- real implementations offer both in one (first in-memory, external memory when needed)
- we split both cases to simplify the code

HJJoinInMem::produce()
  prepare an in-memory hash table
  left.produce()
  right.produce()

Hash-Join (2)

HJJoinInMem::consumeFromLeft(t)
  store t in hash table[$t.a$]

HJJoinInMem::consumeFromRight(t)
  for each t_L in hash table[$t.b$]
    if $p(t_L \circ t)$
      consumer.consume($t_L \circ t$)

- for the combined case consumeFromLeft would check for overflows
- switch to external memory hash join when memory is full

Hash-Join (3)

HJJoinExternal::produce()
  prepare an in-memory spool buffer
  prepare a temp segment for spooling
  define initial partition boundaries
  left.produce()
  flush the buffer if needed
  repartitionLeft()
  right.produce()
  flush the buffer if needed
  joinPartitions()

- left side and right side are partitioned
- recursive re-partition might be needed
- once partitions fit, partitions can be join in-memory

Hash-Join (4)

HJJoinExternal::consumeFromLeft(t)
  if not can spool t to the buffer
    sort the buffer by hash values
    write entries in corresponding partitions
    update partition statistics
    empty the buffer
  spool t into buffer

- left side is materialized in memory
- when memory is full, data is written to corresponding partitions on disk
- here: sort to produce sequential I/O

Hash-Join (5)

```
HJJoinExternal::repartitionLeft()
  for each partition larger than main memory
    if number of different hash values>1
      derive finer partition bounds within the partition
      load partition piecewise into memory
      write out each piece into finer partitions
      replace large partition with finer partitions
    else
      mark the partition as overflow
```

- breaks large partitions into finer partitions
- until the partition fits into main memory
- overflow partitions are a special case

Hash-Join (6)

HJJoinExternal::consumeFromRight(t)
  if not can spool t to the buffer
    sort the buffer by hash values
    write entries in corresponding partitions
    empty the buffer
  spool t into buffer

- materializes, just like the left side
- but can use the proper partition boundaries now

Hash-Join (7)

```
HJJoinExternal::joinPartitions()
  for each partition P
    if P is an overflow partition
      joinPartitionOverflow(P)
    else
      joinPartition(P)
```

- partitions are joined pair-wise
- due to the equal join, join partners can only be found in corresponding partitions
- the overflow case needs some special care

Hash-Join (8)

HJJoinExternal::joinPartition(P)
  load P for the left side into a hash table
  for each t in the right partition P
    for each t_L in hash table[$t.b$]
      if $p(t_L \circ t)$
        consumer.consume($t_L \circ t$)

- P is known to fit for the left side
- brings it to the in-memory case

Hash-Join (9)

HJJoinExternal::joinPartitionOverflow(P)
  for each memory sized chunk of P from left
    load the chunk into a hash table
    for each t in the right partition P
      for each t_L in hash table[$t.b$]
        if $p(t_L \circ t)$
          consumer.consume($t_L \circ t$)

- partition did not fit (identical values)
- similar to a blockwise-nested-loop join
- but uses a hash-table to speed up finding matches

Singleton Join

A singleton-join is a join $e_1 \bowtie^{1J} e_2$

- where $|e_1|$ is guaranteed to be ≤ 1
- relatively common, e.g., after group-by, scalar subqueries, etc.
- nested loop join would work, of course
- but singleton is even simpler (see following slides)

SingletonJoin::produce()
  left.produce()
  right.produce()

SingletonJoin::consumeFromLeft(t)
  $t_L = t$

SingletonJoin::consumeFromRight(t)
  if $p(t_L \circ t')$
    consumer.consume($t_L \circ t$)

Non-Inner Joins

So far we have only consider inner joins. But:

- ⟕, ⟖, ⟗ have to produce non-matching tuples, too
- $\ltimes$, $\rtimes$ have to to produce only matching tuples, without multiplicity
- $\triangleright$, $\triangleleft$ have to to produce only matching tuples, without multiplicity

Similar mechanism, but requires additional bookkeeping.

Non-Inner Joins (2)

Idea: non-inner joins *mark* tuples with a join partner

- outer joins: additional pass, produce non-marker tuples with NULL values
- semi joins: produce only if not marked yet
- anti joins: additional pass, produce only non-marked tuples

Problem: where to store the marker bit?

Non-Inner Joins (3)

Marking the left side is usually simple:

- left tuples in-memory for potential join partners
- reserve a bit in the memory block/hash table/...
- bit can be examined before flushing the memory

Marking the right side is more problematic:

- ok if a right hand tuple is only considered once
- then, store the marker in a register
- but nested loop joins etc. are difficult
- maintain extra data structure (interval compression)

Sort

Sorting is useful for a number of other operations (sm-join, aggregation, duplicate elimination, etc.)

Basic strategy:

1. load chunks of tuples into memory
2. sort in-memory, write out sorted runs
3. merge sorted runs
4. recurse if needed to handle merge fanout

Implementation varies a bit.

Sort (2)

```
Sort::produce()
  input.produce()
  flush memory
  merge partitions
  for each t in merged partitions
    consumer.consume(t)

Sort::consume(t)
  if not enough memory to store t
    flush memory
  materialize t in memory
```

How to implement flush and merge?

Sort (3)

Easy solution for flushing:

- sort the in-memory tuples using quick sort
- write out all of them, release all memory
- fast sort algorithm, simple

More complex solution:

- use heap sort with replacement selection
- write out only when needed
- produces longer runs
- for sorted input: one run
- variable-sized records are difficult to handle

Sort (4)

Merging is usually performed on the fly:

- read all runs in parallel
- retrieve always the smallest element
- tree of losers, or some other priority queue

Problem: what if the number of runs is too large?

- perform a partial merge
- reduces the number of runs
- repeat until merge is feasible

Group By

Aggregation can be implemented in two ways:

Sort based

- input is sorted by group-by attributes
- all tuples within one group are neighboring
- aggregation is largely trivial

Hash based

- aggregate into hash table
- spill to disk if needed
- merge spilled partitions, similar to hash join partitioning

Group By (2)

InMemoryGroupBy::produce()
  initialize an empty hash table
  input.produce()
  for each group t hash table
    consumer.consume(t)

InMemoryGroupBy::consume(t)
  if hash table[$t_{|A}$] exists
    update the group hash table[$t_{|A}$]
  else
    create a new group in hash table[$t_{|A}$]

Variable-length aggregates require some care.

Set Operations

The DBMS also has to offer set operations

- `union` / `union all`
- `intersect` / `intersect all`
- `except` / `except all`

Somewhat similar to joins, but have a very specific behavior.

Set Operations (2)

`union all`

The only trivial set operations

- concatenates both tuple streams
- attribute rename required
- but otherwise nearly no code

Set Operations (3)

`union`

Like a union, but without duplicates

- can be implemented as `union all` followed by duplicate elimination
- $e_1 \cup e_2 \equiv \Gamma_A(e_1 \bar{\cup} e_2)$
- or: directly write into one hash table
  (slightly more efficient, but nearly identical)

Set Operations (4)

`intersect`

Similar to a semi-join

- $e_1 \cap e_2 \approx e_1 \ltimes e_2$
- only true if e_1 is duplicate free
- can be checked during the build phase
- or: use a strategy like for `intersect all`

Set Operations (5)

`intersect all`

Intersection with bag semantics

- defined via characteristics functions
- group by sides into one hash table
- count occurrences on left and right side
- intersect result is minimum of both
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side

Set Operations (6)

`except`

Similar to a anti-join

- $e_1 \setminus e_2 \approx e_1 \triangleright e_2$
- only true if e_1 is duplicate free
- can be checked during the build phase
- or: use a strategy like for `except all`

Set Operations (7)

`except all`

Set difference with bag semantics

- defined via characteristics functions
- group by sides into one hash table
- count occurrences on left and right side
- except result is $max(0, l_c - r_c)$
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side