Introduction

Hash join is a high performance join method
- Supports inner & outer joins on matching columns
- Exploits main memory for efficient row matching
- Best method when no indexes or weak local preds

Terminology
- Build table: the rows that go into the hash table
- Probe table: the rows that match to the hash table
- Key encoding: join columns represented as byte string
- Hashcode: function from byte string to integer
**Overview: from 100,000 meters**

- Encode keys and compute hashcode of encoded keys for build table rows
- Load build table rows into hash table
- Encode keys and compute hashcode of encoded join keys for probe table rows
- Match probe hashcode in hash table and compare encoded keys
- Output matched rows

**Overview: from 50,000 meters**

★ Build table too big for memory

- Partition build table, by hashcode of join keys, into chunks that will fit in memory
- Write build partitions to temporary tables
- Partition probe table, by hashcode of join keys
- Write probe partitions to temporary tables
- Goto 100,000 meters for each partition
Overview: from 10,000 meters

★ Keep some partitions in memory
- Encode and compute hashcode of build join keys
- Partition build table, by hashcode of join keys
- Spill build partitions to build temp tables as needed
- Load rows of remaining partitions into hash table
- Encode and compute hashcode of probe join keys
- If partition in memory, match probe hashcode in hash table and compare encoded keys
- Else spill to probe partition temp table
- Goto 100,000 meters for each spilled partition

Hash Key Encoding

For join predicates \( B.C1 = P.C1 \) AND \( B.C2 = P.C2 \)
we want \( \text{encode}(B.C1,B.C2) = \text{encode}(P.C1, P.C2) \)
so that \( \text{hash}(\text{encode}(B.C1,B.C2)) = \text{hash}(\text{encode}(P.C1, P.C2)) \)

only equal predicates on identical types
since '\abc  ' = 'abc', must strip trailing blanks for VARCHAR keys
ignore rows with NULL key columns (unless outer join probe row)
How many partitions?

- Determine amount of memory
  
  \[
  \text{WANTED} = \min( \text{SORTHEAP}, \text{OPTIMIZER\_BUILD\_SIZE\_ESTIMATE} )
  \]
  
  \[
  \text{ALLOWED} = \text{fcn} ( \text{WANTED}, \text{SORTHEAP\_THRESH}, \text{CURRENT\_USAGE} )
  \]

- Compute number of partitions
  
  \text{GOAL}: Enough partitions in memory for "bin packing" without too many spilled partitions \( (\text{IN\_MEM\_PARTS} = 10) \)

  - Solve for \#PARTS:
    \[
    \text{ALLOWED} = \text{HASHTABLE} + \text{BIT\_FILTER} + \#\text{PARTS} \times (\text{PARTITION\_OVERHEAD}) +
    (\text{OPTIMIZER\_ESTIMATE} / \#\text{PARTS}) \times \text{IN\_MEM\_PARTS} +
    (\#\text{PARTS} - \text{IN\_MEM\_PARTS}) \times \text{SPACE\_PER\_SPILLED}
    \]

  - Sanity checks
    - \#PARTS <= 100
    - imaginary \#PARTS <= \#PARTS = 100

Build Phase

- For each build tuple
  
  - encode join key & compute hashcode
  
  - set bits in bit filter (optimizer option)

  - select partition: \( \text{PART} = \text{hashcode} \mod \#\text{PARTS} \)

  - build hash tuple in \text{tupleBlock}[\text{PART}]

- Memory Full
  
  - spill largest already spilled partition OR largest un-spilled partition

  - insert 4K-byte tupleBlocks into partition's temp table

  - Performance: build directly from buffer pool (if possible)
  
  - SMP: process build tuples in parallel
Pack Memory for Probe Phase

- Determine *winner* and *loser* partitions
  
  - Initially: winners = \{part | partition is unspilled\}
    
  - losers = \{part | partition is spilled\}
  
  - Make some *losers* be *winners* until memory full
    (save room for looser spill buffers)
  
  - Improve packing by (~250) random swaps
    
    - swap types (winner, loser) \(\rightarrow\) (loser, winner)
      
    - (winner, winner, loser) \(\rightarrow\) (loser, loser, winner)
      
    - (winner, loser, loser) \(\rightarrow\) (loser, winner, winner)
    
    - accept swap if feasible and positive I/O savings
      
      \[I/O \text{ savings} = 2 \times (\text{probeSize/buildSize}) \times (\text{winnerSize} - \text{loserSize}) + \text{winnerCopy} + \text{looserCopy}\]
  
- Evict *losers* from memory, read in (partially) spilled *winners*, enter *winners* in hash table
  
- SMP: evict, read, & enter partitions in parallel

Probe Phase

- For each probe tuple
  
  - encode join key & compute hashcode
  
  - if probe tuple from a *winner* partition
    
    - search hash table & compare encoded join keys
    
    - if matched: decode build join key and payload, evaluate 'residual' predicates
    
    - return matched join row (or preserved outer join tuple)
  
  - if probe tuple from a *loser* partition
    
    - test bit filter (optimizer option)
    
    - build hash tuple into `tupleBlock[LOSER]`
    
    - if memory full, spill largest *looser*
  
- Performance: encode key + payload, and compute hashcode directly from buffer pool
  
- SMP: process probe tuples in parallel
Leftovers Phase

- Spill in-memory loser tuple blocks to temp tables
- For each loser partition:
  \( f \) Read in build tuple blocks, enter tuples into hash table
    \( \rightarrow \) won't fit u hash loops algorithm
  \( f \) Read probe tuple blocks. For each probe tuple:
    \( \rightarrow \) search hash table & compare encoded join keys
    \( \rightarrow \) if matched: decode build and probe join key + payload,
      evaluate 'residual' predicates
    \( \rightarrow \) return matched join row (or preserved outer join tuple)

\textbf{SMP:} process loser partitions in parallel

Leftovers Phase: Hash Loops

- Loser build partition is too big for memory
- Repeatedly:
  \( f \) Fill memory with next chunk of build tuples, enter into hash table
  \( f \) Read probe tuples. For each probe tuple:
    \( \rightarrow \) search hash table & compare encoded join keys
    \( \rightarrow \) if matched: decode build and probe join key + payload,
      evaluate 'residual' predicate
    \( \rightarrow \) return matched join row (or preserved outer join tuple)

- Outer join
  \( f \) No match in current build chunk not imply unmatched!
  \( f \) Role Reversal
    \( \rightarrow \) Reverse build and probe roles for hash loops (i.e., read chunks from probe)
    \( \rightarrow \) Mark matched tuples in current probe chunk
    \( \rightarrow \) Return un-marked probe tuples as preserved outer join tuple
Hash Join Performance

Twice as fast as sort + merge join

\( f \) Doesn't preserve order for GROUP BY or ORDER BY

\( f \) Less I/O than sort + merge join

\( f \) when smaller table bigger than SORTHEAP

\( f \) MGJN I/O = OUTER size + INNER size (if both tables are sorted)

\( f \) HSJN I/O = (1 - SORTHEAP/BUILD size) * MGJN I/O

\( f \) when smaller table smaller than SORTHEAP

\( f \) MGJN I/O = BIGGER size (if bigger than SORTHEAP)

\( f \) HSJN I/O = 0

\( f \) Uses more memory for complex queries

\( f \) max of 2 concurrent sorts for merge join queries

\( f \) SORTHEAP per concurrent hash join

Using Hash Join

- **Must** have fixpack 8 or version6.1
- Hash join is enabled by default: `DB2SET db2_hash_join=off`
- Use DBMON:

  GET SNAPSHOT FOR DB ON <database alias>
  - \( f \) Number of hash joins (hash joins executed)
  - \( f \) Number of hash loops (frequent values or SORTHEAPTHRES)
  - \( f \) Number of hash join overflows (build table bigger than SORTHEAP)
  - \( f \) Number of small hash join overflows (bigger SORTHEAP may help)

  GET SNAPSHOT FOR DB MANAGER
  - Hash joins after heap threshold exceeded (requested SORTHEAPs > SORTHEAPTHRESH. Buffers smaller than requested. Can cause hash loops.)