Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Italiano
Português
Русский
Edit links

Hash join

From Wikipedia, the free encyclopedia

The **hash join** is an example of a join algorithm and is used in the implementation of a relational database management system.

The task of a join algorithm is to find, for each distinct value of the join attribute, the set of tuples in each relation which have that value.

Hash joins require an equijoin predicate (a predicate comparing values from one table with values from the other table using the equals operator '=').

Contents [hide]

Classic hash join [edit]

The classic hash join algorithm for an inner join of two relations proceeds as follows:

- First prepare a hash table of the smaller relation. The hash table entries consist of the join attribute and its row. Because the hash table is accessed by applying a hash function to the join attribute, it will be much quicker to find a given join attribute's rows by using this table than by scanning the original relation.
- Once the hash table is built, scan the larger relation and find the relevant rows from the smaller relation by looking in the hash table.

The first phase is usually called the **"build" phase**, while the second is called the **"probe" phase**. Similarly, the join relation on which the hash table is built is called the "build" input, whereas the other input is called the "probe" input. It is like merge join algorithm.

This algorithm is simple, but it requires that the smaller join relation fits into memory, which is sometimes not the case. A simple approach to handling this situation proceeds as follows:

```
1. For each tuple $r$ in the build input $R$
    1. Add $r$ to the in-memory hash table
    2. If the size of the hash table equals the maximum in-memory size:
        1. Scan the probe input $S$, and add matching join tuples to the output relation
        2. Reset the hash table
2. Do a final scan of the probe input $S$ and add the resulting join tuples to the output relation
```

This is essentially the same as the block nested loop join algorithm. This algorithm scans S more times than necessary.

Grace hash join [edit]

A better approach is known as the "grace hash join", after the GRACE database machine for which it was first implemented.

This algorithm avoids rescanning the entire S relation by first partitioning both R and S via a hash function, and writing these partitions out to disk. The algorithm then loads pairs of partitions into memory, builds a hash table for the smaller partitioned relation, and probes the other relation for matches with the current hash table. Because the partitions were formed by hashing on the join key, it must be the case that any join output tuples must belong to the same partition.

It is possible that one or more of the partitions still does not fit into the available memory, in which case the algorithm is recursively applied: an additional orthogonal hash function is chosen to hash the large partition into sub-partitions, which are then processed as before. Since this is expensive, the algorithm tries to reduce the chance that it will occur by forming as many partitions as possible during the initial partitioning phase.

Hybrid hash join [edit]

The hybrid hash join algorithm[1] is a refinement of the grace hash join which takes advantage of more available memory. During the partitioning phase, the hybrid hash join uses the available memory for two purposes:

1. To hold the current output buffer page for each of the k partitions
2. To hold an entire partition in-memory, known as "partition 0"

Because partition 0 is never written to or read from disk, the hybrid hash join typically performs fewer I/O operations than the grace hash join. Note that this algorithm is memory-sensitive, because there are two competing demands for memory (the hash table for partition 0, and the output buffers for the remaining partitions). Choosing too large a hash table might cause the algorithm to recurse because one of the non-zero partitions is too large to fit into memory.

Hash anti-join [edit]

Hash joins can also be evaluated for an anti-join predicate (a predicate selecting values from one table when no related values are found in the other). Depending on the sizes of the tables, different algorithms can be applied:

Hash left anti-join [edit]

- Prepare a hash table for the **NOT IN** side of the join.
- Scan the other table, selecting any rows where the join attribute hashes to an empty entry in the hash table.

This is more efficient when the **NOT IN** table is smaller than the **FROM** table

Hash right anti-join [edit]

- Prepare a hash table for the **FROM** side of the join.
- Scan the **NOT IN** table, removing the corresponding records from the hash table on each hash hit
- Return everything that left in the hash table

This is more efficient when the **NOT IN** table is larger than the **FROM** table

Hash semi-join [edit]

Hash semi-join is used to return the records found in the other table. Unlike plain join, it returns each matching record from the leading table only once, not regarding how many matches are there in the **IN** table.

As with the anti-join, semi-join can also be left and right:

Hash left semi-join [edit]

- Prepare a hash table for the **IN** side of the join.
- Scan the other table, returning any rows that produce a hash hit.

The records are returned right after they produced a hit. The actual records from the hash table are ignored.

This is more efficient when the **IN** table is smaller than the **FROM** table

Hash right semi-join [edit]

- Prepare a hash table for the **FROM** side of the join.
- Scan the **IN** table, returning the corresponding records from the hash table and removing them

With this algorithm, each record from the hash table (that is, **FROM** table) can only be returned once, since it is removed after being returned.

This is more efficient when the **IN** table is larger than the **FROM** table

References [edit]

1. ^ DeWitt, D.J.; Katz, R.; Olken, F.; Shapiro, L.; Stonebraker, M.; Wood, D. (June 1984). "Implementation techniques for main memory database systems". *Proc. ACM SIGMOD Conf* **14** (4): 1–8. doi:10.1145/971697.602261.

External links [edit]

- Hansjörg Zeller; Jim Gray (1990). "An Adaptive Hash Join Algorithm for Multiuser Environments" (PDF). *Proceedings of the 16th VLDB conference* (Brisbane): 186–197. Archived from the original (PDF) on 2012-03-11. Retrieved 2008-09-21.

See also [edit]

Symmetric Hash Join

Categories: Hashing | Join algorithms