Dynamic Memory Hybrid Hash Join

David J. DeWitt and Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison

June 1, 1995

1 Algorithm Overview

Hash joins always operate in three phases:

1. Partition relation A into n buckets by hashing on the join attribute.
2. Partition relation B into n buckets by hashing on the join attribute.
3. Join buckets A_i and B_i for $i = 1, 2, \ldots, n$.

In our external join code, the buckets formed will reside in one of two places: (1) in memory, if there is enough room; or (2) on disk in the Unix file system, if there is not room. In general there will be a combination of (1) and (2), that is, as much as possible will stay in memory, with the remainder of the buckets being written to the file system.

1.1 Which Hash Join?

There are a variety of hash join algorithms to select from including hybrid [DKO+84], Grace [KTMo83], and Adaptive [ZG90]. Grace is the simplist to implement. However, it always writes its buckets to disk. Hybrid and Adaptive attempt to keep the buckets for the inner (smaller) relation in memory if possible. Hybrid depends on having a good estimate of the size of the inner relation and how much memory is available in the split server for joining the tables. Adaptive is probably the best algorithm. It will keep as much of the inner table in memory as possible and does not require as accurate estimates from the optimizer as hybrid. In addition, it can adapt dynamically to changes in the amount of memory available. It is also the most complicated to implement.

Here we describe a new algorithm that combines the best features of Adaptive with the simplicity of Grace. Its detailed design is described in the following section; here we give a very brief overview.

Hybrid hash works by keeping one bucket of the hash table in memory and writing the remaining buckets to disk. If there are few buckets (e.g., if this first bucket is a significant fraction of the inner table) this is a huge benefit. Not only do we avoid writing this bucket to disk; we also avoid writing the corresponding bucket of the outer table to disk, since it can be "streamed" directly past this in memory bucket.

The problem, as we stated above, is that it is difficult to arrange things so that this first bucket is the correct size. To do so you need to know the size of the inner table (difficult if

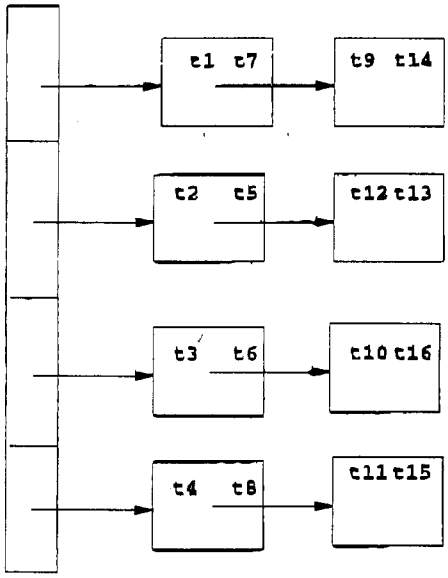

Figure 1: Initial setup for adaptive hash.

it is an intermediate result of a complex query), and even if you know that, the algorithm is sensitive to skew (the hash function may not partition the table into equal sized buckets.)

Our idea to solve this problem is to begin with far more buckets than we will need. Initially, we try to keep all buckets in memory. Figure 1 shows a configuration in which there are four buckets (in reality there would be more, e.g. 100, but that is hard to draw!). Furthermore, the figure assumes that we can fit two tuples to a page, and that we can allocate 8 pages to storing the inner table in the hash join. Each box in the figure is a page.

When we start running out of memory, we begin writing some of the buckets to disk. Figure 2 illustrates how. In the figure, we have allocated 8 pages, all of which are full, and an additional tuple arrives, tuple t17, bound for bucket zero. We need a new page for bucket zero, but have already allocated eight pages to the hash table. So, to make space, we "freeze" the last bucket (bucket 3, if we start numbering at zero.) This means that we write all data that was in the chain for bucket three to disk (in a Unix file.) We retain one page (the shaded page) as a buffer for new tuples inserted into bucket 3; whenever this buffer fills it is written out to disk and cleared.

Later, suppose that enough new tuples have been inserted into buckets zero and one. In this case we need to freeze bucket two. Figure 3 illustrates this case. Now any tuple mapped to buckets two or three will be written to disk (after the corresponding buffer fills.)

If no more tuples arrive (the inner relation is completely processed) then the final state will find all tuples for buckets zero and one in memory, and all tuples for buckets two and three on disk. When the probing relation is processed, tuples for buckets two and three are written to disk, while tuples for buckets zero and one probe an in-memory hash table built out of the tuples of the inner relation that mapped to those buckets.

A final thing to notice is that after processing the inner we have complete knowledge about the size of each of the buckets. This enables the join algorithm to always read in about one memory's worth of data (by reading the appropriate number of buckets.) The next section gives some details of this approach.

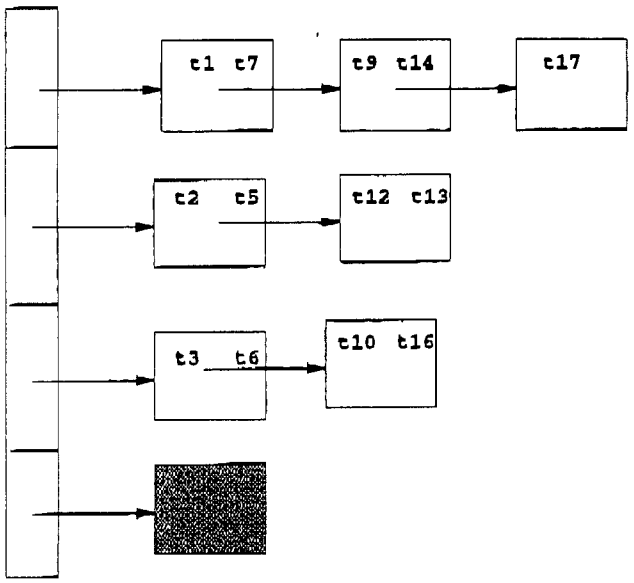

Figure 2: Freezing bucket three.

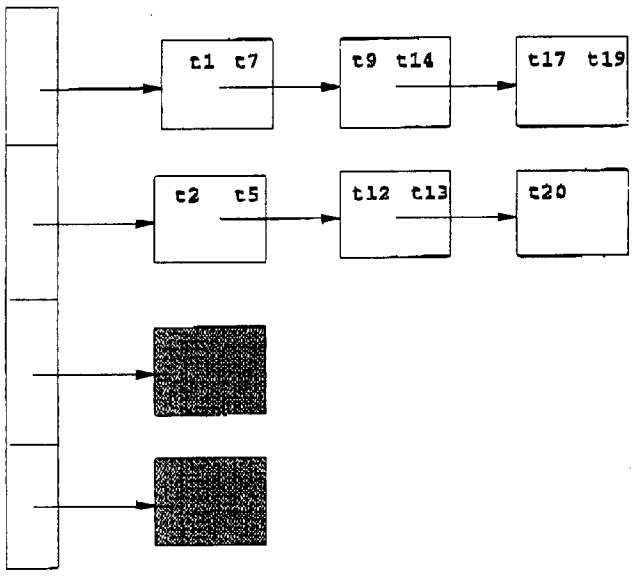

Figure 3: Freezing buckets two and three.

2 Algorithm Details

In this section we describe the partitioning and joining steps in more detail. We assume that the optimizer will pick the smaller of the two relations as the "inner" relation.

In designing these algorithms we had the following goals:

1. The algorithm should keep as much of the inner relation in memory as possible. This is complicated by the fact that we may not know ahead of time how large the inner relation will be, or how much memory is available for buffers. However, the algorithm should never thrash.
2. The algorithm should be robust in the presence of skew. That is, skew may degrade performance somewhat, but it will not cause the algorithm to fail or thrash.

2.1 Data structures

We assume the following variables, constants, and data structures. In the actual implementation, these variables will be attributes of a "hashJoin" structure. There will be one instance of this structure for each concurrently executing join. This structure has been omitted to simplify the presentation.

```
int maxBufs;  // # of page size buffers available to execute
     // the partitioning and joining phases.  Set by the optimizer
     // on a join by join basis

int curBufs; // # of buffers currently in use.  Initialized to 0

int numBuckets; // number of buckets.  Set by the optimizer.  Value
// must be set low enough to avoid running out of file
// descriptors.  A default value of 100 would be reasonable

enum BucketState {empty, expanding, frozen};

typedef struct bucket
{
    BucketState state;  // empty, expanding, or frozen
    int tupCnt;  // number of tuples currently in bucket.
    int pageCnt; // number of pages currently in bucket;
    int resPageCnt; // number of memory resident pages;
    char* firstPage; // pointer to first buffer page in memory
    char* lastPage; // pointer to last buffer page in memory
    File* file; // file descriptor (may be NULL)
    char fileName[MAXNAME];  // name of file for disk resident pages
}
```

2.2 Partitioning the Inner Table

The following two data structures are used while partitioning the inner relation into buckets:

```
bucket innerRelation[numBuckets]; // bucket table for inner relation
int frozen; // number of buckets that have been "frozen".
```

This phase begins by dynamically allocating the innerRelation table. The "state" of each entry is set to empty. The variables `tupCnt`, `pageCnt`, and `resPageCnt` are all set to 0. Also, `firstPage` is set to NULL as is `file`. Finally, `frozen` is set to 0.

As tuples arrive, they get partitioned into one of the `numBuckets` buckets. This is done as follows:

1. Extract join attribute from input buffer.
2. Hash join attribute to produce a hash value. Take `numBuckets` mod hash value to produce a bucket identifier i.

   Finally, increment `innerRelation[i].tupCnt`.
3. Bucket i can be in one of 3 states: `empty`, `frozen`, `expanding`. Next,
   - If (state == empty) perform the following actions:
     - dynamically allocate a new buffer page, setting `firstPage` and `lastPage` appropriately.
     - set `pageCnt = 1; resPageCnt = 1;`
     - increment `curBufs`;
     - If (`curBufs` $\geq$ `MaxBufs` invoke purge();
     - copy tuple from input stream to the buffer page.
   - If (state == expanding)
     - examine buffer page pointed to by `innerRelation[i].lastPage`. If sufficient room exists, copy tuple from input stream to the buffer page
     - otherwise,
       * allocate a new buffer page
       * add page to the end of the linked list of buffer pages for this bucket
       * `pageCnt++; resPageCnt++;`
       * increment `curBufs`;
       * If (`curBufs` $\geq$ `MaxBufs`) invoke purge();
       * copy tuple from input stream to the buffer page.
   - If (`state = frozen`)
     - examine buffer page pointed to by `innerRelation[i].firstPage`; if sufficient room exists, copy tuple from input stream to the buffer page
     - otherwise, write current page to disk file (if file is NULL at this point, an error has occurred); increment pageCnt; copy tuple from input stream to the now empty buffer page.

While the state of a bucket is "expanding", new memory-resident buffer pages are added in a linked list as tuples are added to the bucket. This partitioning process continues until either the entire inner relation has been processed or until `curBufs` $\geq$ `maxBufs`. If the former occurs first, the inner relation will be totally memory resident and no disk I/Os will have occurred. Otherwise, when (`curBufs` $\geq$ `maxBufs`), purge() will be invoked.

purge() picks one bucket to freeze. While there are a number of possible ways to pick the victim (we intend to study these later), initially we will do this by simply incrementing the variable frozen. When a bucket is frozen the following operations are performed:

1. Create a temporary Unix file to hold pages of the bucket.
2. Open the file, saving the file descriptor in the bucket's entry in the innerRelation structure.
3. Write all the memory resident buffer pages for the bucket to its associated Unix file.
4. Release all buffer pages except the first one.
5. set `curBufs` to `curBufs - (resPageCnt-1)`.
6. Adjust `firstPage` and `lastPage` appropriately. Set `resPageCnt` to 1. Mark the remaining page empty.
7. set bucket state to frozen.

The process of "freezing" a bucket releases a bunch of buffer pages for other buckets to use. Once a bucket has been "frozen" the bucket is allowed to use only a single buffer page. As this page fills, it is written to disk.

After a bucket has been frozen, the partitioning process continues until the entire inner relation has been consumed. At this point all buckets i, for $0 \leq i <$ `frozen` -1, are frozen. Buckets j, `frozen` $\leq j <$ `numBuckets`, will be entirely memory resident.

The partitioning of the inner relation finishes with the two following steps:

1. Flush the remaining memory-resident buffer page of each frozen bucket to disk. For each such bucket, release the buffer page to the free list, decrement `curBufs` by 1, close the associated file, set file to NULL, adjust all pointers, etc.
2. From the non-frozen buckets (buckets j, for frozen $\leq j <$ `numBuckets`) form a memory resident hash table (called HashTable and implemented using chained bucket hashing). It may be the case that there are no such buckets. Each bucket is processed in turn, hashing on the join attribute of each tuple in the bucket. The size of the hash table (`HashTblSize`) can be determined by adding the `tupCnt` values of each bucket. No tuples need to be copied. Rather, allocate little hash chain entries consisting of simply a pointer to the tuple and a link to the next entry in the hash chain.

Once the inner relation has been partitioned, we are left:

1. Some frozen partitions completely on disk
2. A memory resident hash table for the remaining buckets (if any)
3. A innerRelation structure describing all the buckets in detail.
4. The value of frozen, which acts as an important dividing line during the next phase of the algorithm.

At this point we can check several things including whether there is skew, whether any of the buckets are bigger than the amount of memory allocated to the join, etc. and take the necessary corrective action.

2.3 Partitioning the Outer Table

As with the inner table, we need an instance of the bucket structure to keep track of the buckets formed during partitioning the outer relation:

```
bucket  outerRelation[numBuckets];  // bucket table for outer relation
```

As before this array is allocated dynamically at the beginning of the phase. As with the inner table, the "state" of each entry is set to `empty`, the variables `tupCnt`, `pageCnt`, and `resPageCnt` are all set to 0, and `firstPage`, `lastPage`, and `file` are set to `NULL`.

As tuples arrive, they get partitioned into one of the `numBuckets` buckets. This is done as follows:

1. Extract the join attribute from the next tuple in the input buffer
2. Hash the join attribute to produce a hash value (JHV). Mod JHV by `numBuckets` to produce a bucket identifier i.
3. If ($i <$ `frozen`) the tuple will be added to a bucket whose corresponding bucket for the outer relation is frozen. The state of the ith bucket of the outer relation will either be `empty` or `frozen`. The following operations are performed next.
   - If (`state = empty`) perform the following actions:
     - dynamically allocate a new buffer page, setting `firstPage` and `lastPage` appropriately.
     - Set `pageCnt = 1`; tt resPageCnt = 1;
     - increment `curBufs`
     - copy tuple from input stream to the buffer page.
     - mark bucket state as `frozen`
     - Create a temporary Unix file to hold pages of the bucket.
     - Open the file, saving the file descriptor

     Notice that there is no check for whether `curBufs > MaxBufs`. We have no choice but to allocate a page to the buffer. If things are really tight, the initial value of MaxBufs should be adjusted appropriately.
   - If (`state == frozen`)
     - examine buffer page pointed to by `outerRelation[i].firstPage` if sufficient room exists, copy tuple from input stream to the buffer page
     - otherwise,
       * write current page to the Unix file associated with the bucket. (if file is NULL at this point, an error has occurred).
       * increment `pageCnt`;
       * copy tuple from input stream to the now empty buffer page.
   - If ($i \geq$ `frozen` and $i <$ `maxBufs`) the tuple can be joined immediately with the inner table. This is done as following.

     (a) First take JHV (see step 2) mod HashTblSize to produce a value j between 0 and HashTblSize - 1.

(b) Search hash chain starting at `HashTable[j]` for tuples from the inner relation with matching join attribute values.

(c) When a matching tuple is found materialize a result tuple from the inner and outer tuples (we are going to need some Navigator help here) and send the result tuple on its merry way (either off to another Split Server or back into the local SQL Server).

Once all the tuples of the outer relation have been processed, flush the memory-resident buffer page of each frozen bucket to disk. For each such bucket, release the buffer page to the free list, decrement `curBufs` by 1, close the associated file, set `file` to NULL, adjust all pointers, etc.

At this point,

(a) buckets 0 to `frozen-1` for both the inner and outer relation are on disk

(b) the join of buckets frozen to `maxBufs` have been completed.

(c) the innerRelation and outerRelation bucket structure remain in memory.

2.4 Joining the Frozen buckets

All that is left to do at this point is to join the frozen buckets. This process operates as follows:

For $i = 0$ to `frozen-1` do

- compare `innerRelation[i].pageCnt` and `outerRelation[i].pageCnt`
- pick the smaller to be the "inner" bucket
- use the `tupCnt` information of the inner bucket to create an appropriately sized hash table
- open the Unix file associated with the inner bucket.
- read the pages of the inner bucket from the file, inserting the tuples into the memory resident hash table by hashing on their join attribute.
- close the inner file
- open the Unix file associated with the outer bucket.
- read the pages of the inner bucket from the file. As each page is read, examine the tuples one at a time, extract the join attribute, hash it, and then probe the hash table for matches. Process result tuples in the normal way.
- close the outer file.
- destroy the inner and outer files

At this point we are done. Release the space occupied by the innerRelation and outerRelation arrays and do general cleanup.

References

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Lenard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1), 1983.

[ZG90] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser envirnoments. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 186–197, August 1990.