Correctness Proofs for the
Gamma Database Machine Architecture

Don Batory * batory@cs.utexas.edu
Bryan Marker * bamarker@cs.utexas.edu

Abstract
The Gamma database machine encodes a classical way in which relational joins are parallelized [2]. In a companion technical report [1], we derived an MDE architecture of Gamma as a series of model-to-model transformations. In this report, we give a proof of correctness for each of these transformations. By starting with a correct architecture (a single join component), and applying correct transformations, the resulting architecture (Gamma’s parallel join architecture) is correct. Our work is an example of a correct-by-construction approach to software design.

1 Introduction
Gamma was (and perhaps still is) the most sophisticated relational database machine built in academia [2]. It was created in the late 1980s and early 1990s without the aid of modern software architectural models. In a companion technical report [1], we presented the architecture used in Gamma to parallelize hash joins. We started with a single hash join component and applied transformations to refine and optimize the implementation of this component into a pipe-and-filter architecture that Gamma used to parallelize joins.

In this report, we focus on the correctness of these transformations. By starting with a correct architecture (a single join component), and applying correct transformations, the resulting architecture (Gamma’s parallel join architecture) is correct. The sequence of transformations that we consider in this paper is in the order in which they were applied in [1].

2 Hash Joins
Equijoins in databases implement the SQL query in Figure 1a where C is the attribute that is shared between relations A and B. We denote equijoins by A * B. If the relations A and B are extended to include a derived field HC = hash(C),

*Department of Computer Science, The University of Texas at Austin, Austin, Texas 78712
this query can be equivalently expressed as Figure 1b. Note that this derived field adds no additional information, only the hash value of an existing join key.

```
select <fields>
from A, B
where A.C = B.C
```

select <fields>
from A, B
where A.C = B.C and
A.HC = B.HC

(a) Join of Relations A and B over C     (b) Hash Join Query

Figure 1: Equivalent Equijoin Queries

There are two intuitive properties that make the queries of Figure 1a-b equivalent [3]:

Property 2.1. If (A.C = B.C) is true, then (A.HC = B.HC) is true.

Property 2.1 states that the hashes of two equal keys must be equal. If the query of Figure 1a returns a tuple, so will the query of Figure 1b.

Property 2.2. If (A.HC = B.HC) is false, then (A.C = B.C) is false.

Property 2.2 states that if two hash keys are not equal, then their keys are not equal.

3 Implementing Hash Join With Bloom Filters

To parallelize a hash join, we first refine it to an implementation that uses Bloom filters. Figure 2 shows the implementation using a graphical, pipe-and-filter representation.

To prove the correctness of this refinement, note that the same A stream is read by HJOIN in the refinement because BLOOM does not affect it. A different stream, called B’, is read by HJOIN, though. The tuples of B’ are a subset of those of B. We will show that none of the omitted tuples contribute to HJOIN’s output. This proves that HJOIN(A,B’) = HJOIN(A,B).

BLOOM(A) has two outputs: the original stream A (meaning that A is read by BLOOM and is output unchanged) and M, the set of HC values of A. Note $M = \prod_{HC} A$ is the relational projection of A over HC.

BFLITER forms the B’ stream by removing all tuples in B whose HC values are not in M. If tuple $tB \in B'$, then $tB.HC \in M$. This is the semijoin of M and B over HC.

Let $t$ be tuple of HJOIN(A,B) where $tA$ and $tB$ are the tuples of A and B that are joined to produce t. We know that $(tA.C = tB.C)$ and $(tA.HC = tB.HC)$.

Theorem 3.1. $HJOIN(A,B) = HJOIN(A,B')$
Proof by contradiction with two cases.

Case 1: Suppose there is a tuple $t$ output by $HJOIN(A,B)$ that does not belong to $HJOIN(A,B')$. This would only happen if $tB$ is eliminated by $BFILTER$, meaning that $tB.HC \notin M$. We know that $tA.HC \in M$ because $M$ is constructed from all $tA.HC$ values. Furthermore, we know $tA.HC = tB.HC$ because $t \in HJOIN(A,B)$. This is a contradiction:

$$tA.HC \in M \land (tA.HC = tB.HC) \implies tB.HC \in M$$

Case 2: Suppose there is a tuple $t$ that is output by $HJOIN(A,B')$ that does not belong to $HJOIN(A,B)$. Recall the distributivity of joins over unions requires that the join of $X$ with $(Y \cup Z)$ must contain $X * Y$ [3]:

$$X * (Y \cup Z) = X * Y \cup X * Z$$

Since $B' \subseteq B$, we know that $HJOIN(A,B') \subseteq HJOIN(A,B)$. Thus, there is no tuple $t \in HJOIN(A,B')$ that is not in $HJOIN(A,B)$. □

4 Parallelizing Refinement of BLOOM

Now that the equivalence has been proven for $HJOIN$ and its refinement using BLOOM, $BFILTER$, and $HJOIN$, we can parallelize each of these components and prove their equivalence. To do so, we need to add yet another derived hash field $SC$, called split-code, which is a hash of the join key $C$. $SC$ differs from $HC$ in that $SC$ has a very limited range of values, namely $1 \ldots n$ for small $n$, whereas $HC$ could be 36-bit integers (for example). Further, we assume $SC$
Figure 3: Parallelizing Refinement of BLOOM

and HC are related: there is a function H such that \( H(HC) = SC \). So given an HC value we can derive its corresponding SC value.

First, we prove the equivalence of the parallelization of BLOOM, shown in Figure 3. Here, we split stream A by values in SC, that is A is split into substreams \( A_1 ... A_n \). We prove that the union of the hash split of A is A. Then we prove that the union of the \( M_i \) streams, the HC values of the \( A_i \) streams, is M, the output of the Bloom filter.

4.1 Equivalence of A and its Hash Split

HSPLIT is defined to output \( A_i \), the result of the query in Figure 4 for \( i \in 1...n \). Furthermore, BLOOM is defined to output the same \( A_i \) it receives as input and MERGE takes the union \( A_1 \cup ... \cup A_n \), which is equivalent to A because the set \( 1...n \) includes all hash values stored in A.SC. Therefore, this refinement of BLOOM outputs the same A stream as the input.

```
select *
from A
where SC=i
```

Figure 4: \( A_i \) Output by HSPLIT

4.2 Equivalence of M and its Hash Split

Let the Bloom filter of A be M: \( M = BLOOM(A) \). We know that the Bloom filter of A is the set of all of its HC hash values: \( M = \prod_{HC}(A) \).
To prove the equivalence of M and the merged values of the Bloom filter applied to each $A_i$, we use the following identity from relational algebra that states that the projection of a union is the union of projections [3]:

$$\prod_{(X)} (Y) = \prod_{(X)} (Y_1 \cup Y_2) = \prod_{(X)} (Y_1) \cup \prod_{(X)} (Y_2)$$

It follows:

$$BLOOMM(A) = \prod_{(HC)} (A) \quad \text{definition of } BLOOMM$$

$$= \prod_{(HC)} (A_1 \cup \ldots \cup A_n) \quad A = A_1 \cup \ldots \cup A_n$$

$$= \prod_{(HC)} (A_1) \ldots \cup \prod_{(HC)} (A_n) \quad \text{relational algebra identity}$$

$$= BLOOMM(A_1) \cup \ldots \cup BLOOMM(A_n) \quad \text{definition of } BLOOMM$$

Thus, MMERGE, which outputs $BLOOMM(A_1) \cup \ldots \cup BLOOMM(A_n)$, produces the same M as BLOOMM(A), so this refinement of BLOOM is correct.

### 4.3 Implementation Note

Now, we need to briefly present a note on implementation. Consider the following figure which shows a class M (with a getM()) method and a subclass MM, also with a getM() method and an additional getM(i) method. M is a bit map, which encodes a set of hash values; getM() returns this bitmap. Subclass MM stores a set of sets of hash values (namely, it stores $M_1 \ldots M_n$). Its getM() method returns the bitmap of $M_1 \cup \ldots \cup M_n$. Its get(i) method returns $M_i$. The output of MMERGE is an object of type MM, which because of polymorphism is indistinguishable from an object of type M.

Figure 5: The M and MM Class Hierarchy

### 5 Parallelizing Refinement of BFILTER

Similar to the previous section, we prove the equivalence of BFILTER and its parallelizing refinement, shown in Figure 6.
Figure 6: Parallelizing Refinement of BFILTER

\( B' = BFILTER(B, M) \) outputs the result of Figure 7a and \( B'_i = BFILTER(B_i, M_i) \) outputs the result of Figure 7b. MSPLIT takes \( M \) as input and outputs all \( M_i \), the results of Figure 7c, while HSPLIT outputs all \( B_i \), the results of Figure 7d. These components enable parallelization across all hash values.

To prove the equivalence of the refinement of BFILTER, we need to show that the result of MERGE is the same as the result of BFILTER, \( B' \):

\[
B' = B'_1 \cup \ldots \cup B'_n
\]

We know \( B'_i = BFILTER(B_i, M_i) = B_i \ast M_i \) and \( B' = BFILTER(B, M) = B \ast M \).

First, we need to prove the identity:

\[
B_i \ast M_i = B_i \ast (M_1 \cup \ldots \cup M_n) = B_i \ast M
\]

This follows from the observation that \( B_i \ast M_k = \emptyset \) when \( i \neq k \). The query for \( B_i \ast M_k \) is shown in Figure 8. Note that \( H(B_i.HC) = i \) for all tuples in \( B_i \) and \( H(M_k.HC) = k \) for all tuples in \( H_k \). When \( i \neq k \), \( H(B_i.HC) \) never equals \( H(M_k.HC) \), so we have \( B_i \ast M_k = \emptyset \). (Note: \( H(B_i.HC) = B_i.SC \), and \( H(M_k.HC) \) is the SC value of \( M_k.HC \)).

Now, we can prove the equivalence of this refinement:

\[
\begin{align*}
B_1 \cup \ldots \cup B_n' &= B_1 \ast M_1 \cup \ldots \cup B_n \ast M_n \\
&= B_1 \ast M \cup \ldots \cup B_n \ast M \\
&= (B_1 \cup \ldots \cup B_n) \ast M \\
&= B' \ast M \\
&= B'
\end{align*}
\]

definition of \( B'_i = BFILTER(B_i, M_i) \)
from above identity
relational algebra identity
definition of \( B' = BFILTER(B, M) \)
select * from B, M
where B.HC = M.HC
(a) \( B' = BFILTER(B, M) \)

select * from Bi, Mi
where Bi.HC = Mi.HC
(b) \( B'_i = BFILTER(B_i, M_i) \)

select * from M
where H(HC) = i
(c) \( M_i \) output from MSPLIT

select * from B
where B.SC = i
(d) \( B_i \) output from HSPLIT

Figure 7: Component queries for the BFILTER refinement

select *
from Bi, Mk
where Bi.HC = Mk.HC
Figure 8: \( B_i \ast M_k \)

Finally, we need to show that the output of MSPLIT(M) is indeed \( M_1 \ldots M_n \).
Recall Figure 5, where we noted that the output of MMERGE is an object of type MM, which is the object that MSPLIT takes as a parameter. By invoking method getM(i) for \( i \in 1 \ldots m \), MSPLIT(M) produces \( M_1 \ldots M_n \) trivially.

6 Parallelizing Refinement of HJOIN

Now, we consider the correctness of the parallelizing refinement of HJOIN, shown in Figure 9. We need to show:

\[ A_1 \ast B_1 \cup \ldots \cup A_n \ast B_n = A \ast B \]

To that end, we introduce the following:

**Theorem 6.1.** \( i \neq k \implies A_i \ast B_k = \emptyset \)

\( A_i, B_i, \) and \( A_i \ast B_k \) are the results of queries in Figures 10-a-c, respectively. Note that \( A_i.SC = i \) for all \( A_i \) and \( B_k.SC = k \) for all \( B_k \). It then follows that \( i \neq k \implies A_i \ast B_k = \emptyset \). HSPLIT forms \( A = A_1 \cup \ldots \cup A_n \) and \( B = B_1 \cup \ldots \cup B_n \). Therefore, we have:

\[
A \ast B = (A_1 \cup \ldots \cup A_n) \ast (B_1 \cup \ldots \cup B_n)
\]

substitution using above

\[
= A_1 \ast B_1 \cup A_1 \ast B_2 \cup \ldots \cup A_n \ast B_n
\]

distributivity of \( \ast \) over \( \cup \)

\[
= A_1 \ast B_1 \cup A_2 \ast B_2 \cup \ldots \cup A_n \ast B_n
\]

simplification using Theorem 6.1

\[ \square \]
Figure 9: Parallelizing Refinement of HJOIN

\[
\begin{align*}
\text{select } * & \quad \text{select } * & \quad \text{select } * \\
\text{from } A & \quad \text{from } B & \quad \text{from } A_i, B_k \\
\text{where } SC=\text{i} & \quad \text{where } SC=\text{k} & \quad \text{where } A_i.C = B_k.C \\
(a) \ A_i & \quad (b) \ B_k & \quad (c) \ A_i \ast B_k
\end{align*}
\]

Figure 10: Queries for HJOIN Refinement

7 Optimizations

Figure 11 composes all refinements (transformations) that we have considered so far. We now apply optimizations (transformations) to eliminate unnecessary computations.

The first optimization, shown in Figure 12, removes the operations MERGE and HSPLIT. Streams \(A_1 \ldots A_n\) were created by HSPLIT. Merging them into a single stream and HSPLITting this stream in exactly the same manner as before is the identity mapping (i.e., MERGE and HSPLIT are inverses of each other for the input \(A_1 \ldots A_n\)). So the correctness of this optimization is trivially true. Note that this optimization is applied twice in Figure 11 once for stream \(A\) and once for stream \(B'\).

For the optimization of Figure 13, which also removes inefficient inverse operations, we need to prove that MMERGE and MSPLIT are indeed inverse operations.

This is trivially so. The output of MMERGE is an MM object, which remembers each \(M_1 \ldots M_n\). This MM object is input into MSPLIT, which invokes the getM(i) method to recover each \(M_i\). MMERGE and MSPLIT are thus inverses of each other.

Figure 14 shows the optimized design that expresses Gamma’s parallelization of hash joins.
8 Cascading Joins

There is one final optimization which arises when the output of one join becomes the input of another (Figure 15a). Figure 15b reveals the partial internals of HJOIN architectures showing that a MERGE produces a single stream (e.g. C or E) which is immediately HSPLIT into smaller substreams. Figure 15c shows the optimization that we want to achieve – a swapping of HSPLIT and MERGE. This optimization eliminates a serialization bottleneck [2].

In general (as in Figure 15c), a set of substreams $A_1...A_n$ is merged into a single stream $A$, and then $A$ is split into substreams $B_1...B_k$. Let $r_i$ denote a record of substream $A_i$. Record $r_i$ is mapped to substream $B_j$, where $\text{hash}(r_i)=j$. Note this mapping is independent of whether $r_i$ is first merged into a single stream $A$, and then routed to substream $B_j$, or whether it is routed to $B_j$ and then merged with other records targeted for $B_j$. Thus MERGEing substreams and then HSPLITting routes records in the same way as HSPLITting and then MERGEing.

9 Acknowledgements

Bryan Marker was sponsored by a fellowship from Sandia National Laboratories. Batory is supported by the NSF’s Science of Design Project CCF 0724979.
Figure 13: MMERGE and MSPLIT removal optimization

Figure 14: Gamma’s Parallelization of Hash Join

References


Figure 15: Cascading Join Optimizations