Exploiting Asynchronous IO using the Asynchronous Iterator Model

Suresh Iyengar * S. Sudarshan Santosh Kumar † Raja Agrawal ‡

supartha@microsoft.com sudarsha@cse.iitb.ac.in santosh@guruji.com raja.agrawal@sap.com

IIT Bombay
India

Abstract

Asynchronous IO (AIO) allows a process to continue to do other work while an IO operation initiated earlier completes. AIO allows a large number of random IO operations to be issued at once, allowing the disk subsystem to order access to data on disk, reducing average seek times considerably, as well as allowing much better utilization of disks in a multi-disk RAID environments where reads can be done in parallel across disks.

In this paper we address the issue of how to extend a database query execution engine to exploit asynchronous IO. To best exploit AIO, we propose a new iterator model called the Asynchronous Iterator Model, where a *getnext()* call on an operator can return a status LATER instead of blocking on an IO, permitting other actions to be initiated while an IO is pending. We show how to modify the implementation of Index Nested Loop (INL) join by issuing asynchronous requests to a batch of tuple ids. We have prototyped the asynchronous iterator model for INL joins on the PostgreSQL database system, and present performance results that clearly indicate the benefits to be had from exploiting AIO.

1 Introduction

The traditional approach to IO processing, namely synchronous IO, blocks the process issuing an IO operation until the operation has completed. As a result, for many queries that require multiple random IO operations, the CPU is idle most of the time waiting for IO operations to complete; and in environments with multiple disks, all but one of the disks would correspondingly be idle. Current versions of operating systems however support asynchronous IO, which allows a process to continue to do other work while an IO operation initiated earlier completes. Asynchronous IO (AIO) allows overlap of CPU and IO processing; it can also allow a large number of IO operations to be issued at once, allowing the disk subsystem to order access to data on disk, reducing seek times considerably. Such an approach allows much higher utilization of resources, and is particularly important in multi-disk RAID environments where reads can be done in parallel across disks.

As noted by Graefe [4], asynchronous IO can be particularly useful for indexed nested loops join. Indexed nested loops joins are particularly useful in situations where only a few tuples of the inner relation are accessed either because only a small number of tuples are present in the outer relation (after selections are applied), or because only a small number of tuples are fetched from the query result. The latter situation is common when results are presented to a user, who only browses through a few results. A similar observation was made earlier in the context of dependent joins by Goldman and Widom [2], and for index lookups by Raman et al. [8].

In this paper we address the issue of how to extend a database execution engine to exploit asynchronous IO. Our contributions are as follows:

1. We extend the iterator model, which is widely used for implementing relational operations, to better exploit AIO. Specifically, in a situation where a getnext() operation would block waiting for an IO to complete, we allow the operation to return a special status LATER, indicating that it is waiting for an IO operation to complete. Thus, even if a part of the plan tree is blocked on an IO, another part of the tree can continue to make progress. The overall plan tree will never block unless all subparts are blocked.

---

* Currently with Microsoft IDC, Hyderabad
† Currently with Guruji.com
‡ Currently with SAP

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17–19, 2008

2. We extend the Indexed Nested Loops implementation to support asynchronous IO. The implementation reads multiple outer tuples, accesses the index to find matching inner tupleids, and issues asynchronous IO operations on these matching inner tupleids. In contrast, a conventional implementation would block on inner tuple accesses for each outer tuple fetched.

3. We have implemented our model in the PostgreSQL system. Our implementation is well integrated with existing operations, allowing asynchronous INL to execute as part of a query plan, even if other parts of the query plan cannot handle the asynchronous iterator model (i.e. do not recognize the return status LATER).

4. We present a performance study that shows the benefits of asynchronous IO based on our model, using data and some queries from the TPC-H benchmark.

Our proposed asynchronous iterator model can also be applied in the context of asynchronous execution of Web service invocations. In situations where a Web service is invoked multiple times, and possibly multiple Web services must be accessed to process a request, asynchronous access can reduce latency greatly. We believe our model shows promise in this area, which is becoming increasingly important.

The rest of the paper is organized as follows. Section 2 describes related work. We discuss the asynchronous IO interface in Section 3. We present the Asynchronous Iterator Model in Section 4. Section 5 describes Asynchronous Index Nested Loop Join in detail. Section 6 briefly outlines how other operations can benefit from asynchronous IO. Section 7 describes how the PostgreSQL system was modified to support asynchronous IO. Section 8 describes an experimental evaluation of the benefits of asynchronous IO as implemented in PostgreSQL. Section 9 concludes the paper and outlines future work.

2 Related Work

Asynchronous IO is certainly not a new idea, but only recently have operating system kernels such as the Linux kernel, started supporting AIO. There has been very little published literature on the topic of exploiting AIO in databases. Graefe [4] describes a *generalized spool iterator* operator, which prefetches multiple outer tuples into a buffer, and issues AIO on corresponding inner tuples. In a query plan, the spool iterator operator is introduced as a child of an indexed nested loops join operation. The indexed nested loops operation is not modified, and continues to issue synchronous IO operations; the AIO performed earlier basically prefetches data, which reduces the probability of the synchronous IO blocking, and allows the disk subsystem to optimize fetching of data. The buffer can be refilled when it is empty (batch mode), or each time a record is removed from the buffer on completion of an asynchronous request (sliding window mode). Graefe also mentions that Microsoft SQL Server and IBM DB2 keep a fixed set of unresolved IO requests and IO is started concurrently for all these elements. Graefe [4] does not provide any experimental results. In contrast to the results in [4], our asynchronous iterator model allows entire subplans to be non-blocking.

Oracle also uses asynchronous IO, which is a default setting if the OS provides AIO capabilities [5] [6]. AIO is also used in Microsoft SQL Server and DB2 [7]. There is not much documentation publicly available on how these systems exploit AIO, and whatever documentation is available does not describe whether or how AIO is used in query processing (although there is mention of use of AIO for writing log records).

Asynchronous iteration for evaluating queries that combine database and web data is described by Goldman and Widom [2]. Their implementation focuses on dependent joins, which are conceptually similar to indexed nested loops joins, but can access external/web data. In their asynchronous implementation of a dependent join, the join returns a place holder tuple even before the equivalent of index fetch completes. A parent operator, called ReqSync, captures these place holder tuples, and only when matching index fetch results are found are these passed on to a parent operator. They discuss issues in optimal placement of the ReqSync operator. However,

1. They only report results for the case where the dependent join issues web queries, not disk IO, although their techniques are conceptually applicable even for disk IO. Their implementation is on an academic prototype database. In contrast, we report results on disk IO, on a modified version of a widely used database system, PostgreSQL.

2. Our asynchronous model supports the ability to not block, by returning a LATER status. Using this feature, we can make all operators non-blocking, allowing a single thread to asynchronously run different parts of a query plan even if there are intervening operators that would be blocking in their model. Their model does not support this feature.

Work on the Eddies framework for query processing [1] is also related. In this framework, the most closely work is that of Raman et al. [8], who describe an asynchronous approach to index lookup, modeled on [2], but tailored to the Eddies framework. The Eddies framework was designed to handle arbitrary delays in accessing data over a network, allowing processing to proceed as much as possible with data available at any point in time. Asynchronous index access fits

API function	Description
aio_read	Request an AIO read operation
aio_error	Check the status of an AIO request
aio_return	Get the return status of a completed AIO request
aio_write	Request an asynchronous write operation
aio_suspend	Suspend calling process until AIO request/s have completed/failed
aio_cancel	Cancel an AIO request
lio_listio	Initiate a list of AIO operations

Table 1: AIO basic functions

into this model naturally. However, the query processing model of Eddies is rather different from normal database query processing, and as far as we are aware is not in production use. In contrast, our work is based on the iterator model for query processing, which is used in all database systems we are aware of.

3 Asynchronous IO Interface

Asynchronous IO is a standard feature of Linux 2.6 and later kernels, and includes functions to initiate an IO operation (for read or write), to check AIO status, and to cancel a request. The functions are listed in Table 1.

The function *aio_read* is similar to conventional read operation, except that it returns immediately after issuing a request to AIO library. The function *aio_error* can be used to keep track of an IO request in a polling loop. The function *lio_listio* allows a single call to initiate multiple AIO requests. This involves only one context-switch to the kernel mode for initiating AIO for a number of blocks at once, as against one context-switch per block if we use *aio_read*.

Since an asynchronous IO call returns immediately after placing an IO request, a method is required for signaling IO completion. There are in fact three main notification models for handling asynchronous IO completion.

1. Signal-based handler: A signal is generated on IO completion which calls a completion handler function which continues the proceedings.

2. Callback using interrupts: When an IO is complete, an interrupt is generated which calls a handler function. One of the main problems associated with this approach (as also with the signal-based approach), in addition to OS overheads, is the concurrent access to shared data structures such as IO queues by the main query processing thread and the handler function. This makes it necessary to protect the data structures using semaphores or other mutual exclusion method. These overheads together affect the performance of this model in the context of asynchronous disk IO, as shown by the results in our performance study.

3. Polling: The requests are stored in a pending queue, and the program polls the queue periodically to check for IO completion. This method avoids the context switch and synchronization overheads of the other methods. A potential disadvantage of this method is that it can waste CPU cycles due to repeated polling. However our experimental results show that polling performs better than the interrupt-based function callback. Details of how polling is integrated with the asynchronous iterator model implementation are described in Section 7.3.

```
1  while(true){

      Fetch next result tuple by calling getNext
           on rootnode of query plan

6     If return status is SUCCESS
         process tuple e.g. display the tuple
      Else
          If return status is LATER
           continue;
11      Else if return status is END_OF_INPUT
                break;
   }
```

Figure 1: Execution loop for root node

4 Asynchronous Iterator Model

Query processing based on the iterator model is very widely used in database system implementations. Iterators implement a demand-driven pull model [3]. Whenever a node requires a tuple, it invokes the getnext() function of it's child node. Each child node produces the next tuple in its output sequence every time it is called. This tuple is returned to the parent node. Bottom level nodes usually perform operations such as sequential scans or index scans. Upper level nodes are usually join nodes or other operator nodes such as sort or aggregate.

The iterator model is inherently blocking. For example, a join operator cannot return a tuple unless it gets a tuple both from the outer and the inner relation and the tuples satisfy the join predicate. Similarly a sequential scan will not return any tuple until the block containing the next tuple is in memory. Such blocking is not desirable, since it does not allow other work to go on in parallel when the IO operation is blocked.

```
typedef struct NestLoopState
{
  .........
  NestLoopOuterTuple
        nl_outerTuples[MAX_NLOUTERSLOTS];
  bool endOfOuter;

  struct queue *workqueue; // outer slots with
       AIO issued for matching inners
  struct queue *freequeue; // free outer slots

  int numTids;
  struct aiocb **areqs_p[MAX_INNER_TIDS]; //
       For List AIO

  BufQueue pending_req; //pending I/O requests

} NestLoopState;
```

Figure 2: Additions to NestLoopState

To avoid the blocking problem, we extend the iterator model to better support asynchronous IO. If an operator is unable to generate any output with only memory resident data, it can issue an AIO request which will return immediately without blocking.

Our asynchronous iterator model extends the basic iterator model by allowing a node to return a status of LATER to the parent, instead of blocking waiting for IO completion.. The implementation of each operator can decide what to do if it gets a LATER status for one of its input. For example, an implementation could perform other work, such as fetching data from another input, while waiting for the asynchronous IO to complete. Alternatively, an operator could simply return a LATER status to its parent node, or it can even just block, waiting for the child operator to provide a tuple.. The return status "LATER" can cascade all the way to the root node of the query plan. When the root node of a query plan receives a LATER return status, it can repeatedly try to get a tuple from the relevant input, until it succeeds. The execution loop for root node is shown in Figure 1.

We discuss the asynchronous extensions for the various relational operations in Sections 5 and 6. We also note that our description as well as implementation is based on PostgreSQL, although the ideas are applicable to any database.

5 Asynchronous Index Nested Loops Join

In this section, we discuss the additions we have made to the state of INL nodes and asynchronous versions of file and buffer operations. We also present the basic framework for an asynchronous INL iterator.

5.1 State of an INL node

The original state of a nested loop join mainly consists of left and right subplans and qualifier lists. We have modified this state for our iterator model. Our main objective is to operate on a a batch of tuples in each iteration and issue list AIO requests for matching inner tids. The additions we have made to the state are illustrated in Figure 2.

- Each INL node maintains an array of outer tuples. With each outer tuple, we maintain a queue of matching inner TIDs.
- Those outer slots which are not used are maintained in the freequeue. The outer slots which already have AIO issued for their matching inner TIDS are maintained in the work queue.
- Each INL node also maintains the number of inner tuples in numTids for which AIO is issued. This number is decremented each time an inner tuple AIO completes, and the tuple has been retrieved and joined with the corresponding outer tuple.
- An array is maintained to collect requests for each call to list AIO. Also, each INL node records all its pending IO requests in a queue, to poll for completion of AIO requests made from that node. BufReq is a structure which holds a single pending AIO request; it contains the buffer address in to which the data will be copied on IO completion; a set of such pending requests is stored in a BufQueue structure, pending_req, associated with the INL node.

5.2 Asynchronous INL Iterator

We divide the INL iteration into two stages; the first stage fetches tuples and issues AIO requests, while the second stage checks for AIO completion, and joins the returned tuples. We discuss these in detail below.

5.2.1 Fetching outer tuples and issuing AIO requests

Figure 3 shows the first part of the asynchronous indexed nested loops join code, which deals with fetching outer tuples and issuing asynchronous IO requests for inner tuples. We allow up to MAX_INNER_TIDS outstanding inner tuple AIO requests. This logic takes care of the case where an outer tuple matches a large number of inner tuples. In our experiments, we set this to maximum value of 200 requests made at a time. We fetch an outer tuple and copy it into a free outer slot. For each outer tuple, we fetch the tuple ids of matching inner tuples from an index scan incrementing the numTids counter each time. If an INL node receives a LATER status when it tries to fetch an outer tuple, it implies that we have no outer tuples to be processed currently, and we skip the first part of iteration.

An important case to handle here is the case of multi-level INL joins. Since, we return immediately from an IO request, at the end of iteration we might

```
Tuple AsyncIndexNestLoopJoin(NestLoopState
    state)
{

fetch_outerbatch:
 Scale BATCH_SIZE by a factor of 2 in each
     iteration , starting from 1, up to
     MAX_INNER_TIDS
  if(node->numTids <= BATCH_SIZE  && !node->
     endOfOuter){
    while node->numTids <= BATCH_SIZE {
       Dequeue an outer slot T from free queue
       status = Fetch an outer tuple and copy
           into T
       if status is LATER{
           Enqueue T back into free queue
           break;
       }
       if outer tuple is NULL
           node->endIfOuter=1 ;
       for each inner tuple tid matching the
           outer tuple {
           Place inner tuple tid in pending
               inner I/O list
           increment node->numTids
           if node->numTids >= BATCH_SIZE
             break; /* Restart this for loop
                  on next call , skipping
                  earlier code; code details
                  omitted */
       }
       Enqueue the outer tuple slot in work
           queue
       Issue asynchronous read request on all
           the inner tuple tids in the list
       if an outer tuple doesn't have any
           matching inners
          Reuse  the slot

perform_join:
       ......
}
```

Figure 3: Asynchronous INL Iterator : First stage

not have any tuples to join. In this case, we set result status to LATER and return a NULL. Whenever an INL node receives a LATER status from its child, it stops fetching outer tuples and skips to the second stage of the iteration. Also, when we have BATCH_SIZE number of inner tuple ids, we skip to the second stage. Section 7.4 discusses how BATCH_SIZE is varied, up to a maximum value of MAX_INNER_TIDS. If the status is SUCCESS with result as NULL, it signals that no more tuples can be fetched from that child node.

In our implementation since we have not made all the operators asynchronous, we use a status *parentcanhandleLATER* which indicates to the node that its parent node can handle LATER status or not. If it cannot handle LATER status, we do not pass a tuple to it until we get an actual one. The *parentcanhandleLATER* status is set by the parent node for each of its child node during initialization. Although currently not implemented, Async INL and other asynchronous operators must be able to return tuples in order when required by a parent operator such as a merge join which depends on tuples being produced in order.

Each matching inner TID is enqueued in the outer tuple's tid queue. We get the relation and page number for each TID and issue an asynchronous buffer read operation. This buffer read operation is discussed in more detail in Section 7.2. This operation allocates a buffer for the data to be read in and calls the File Read function. In this function, we do not issue an IO request, we just enqueue the request in node's AIO array and in the pending request queue of INL node. Also, we dequeue the outerTuple slot from freequeue and enqueue that in the workqueue. We issue a List AIO operation for all these requests at the end of each iteration. This involves only one context switch for a batch of read requests as against one per conventional read operation.

Also, we start the next iteration of fetching outer tuples as soon as the number of inner tuples that have been joined since the last fetch exceeds some fraction of BATCH_SIZE, rather than as soon as a single inner tuple is joined. This reduces the number of List AIO calls, and increases disk and CPU utilization. In our experiments, we set this value to 10%.

5.2.2 Joining tuples and polling

Figure 4 shows the second part of the asynchronous indexed nested loops join code, which deals with matching returned tuples with pending requests. In the second part of the iteration, we start processing the work queue. For each outer tuple in the workqueue, we have a queue of TIDs for which AIO requests have been issued. For these TIDs we check whether the IO request is complete. If the desired data is in the buffer, then at this point we have an outer tuple and an inner tuple to join. We join the tuples and output the result. If we have joined all the matching inner tuples for a given outer tuple, then we dequeue that slot from the workqueue and enqueue it in freequeue. Else we continue the loop, looking for completed AIO requests. As a heuristic to avoid spending too much time in this check, we terminate the loop if AIO has not completed for any of the first K tuples in the queue; in our experiments, we set the value K to 10% of MAX_INNER_TIDS.

If there are no matching inner tuples in memory at this point, we first check if there are any outer tuples left either in the work queue, or to be retrieved from the outer input; if not, the function signals end of input (by setting tupStat to SUCCESS and returning a null tuple).

Otherwise, there are still outer tuples to be processed, but no results available to return currently, so the return status is set to LATER and the result tuple to NULL. Also, at this point, we poll for AIO completion for the requests enqueued in the node's pending queue. If any request is complete, then we tag that

```
1  perform_join:
       for every outer tuple in workqueue {
          Check whether any of its matching
              inners present in memory
          if present {
              Remove that matching inner from
                  pending inners list
6             Perform join with corresponding
                  outer tuple, and add to
                  result
                 /* Note: above step checks
                     join qualifiers as well as
                      other qualifiers for
                     inner tuples */
              decrement node->numTids
          }
       }
11
       if we are done joining all the inner
           tuples of a particular outer tuple,
              Dequeue that particular slot from
                  the work queue

       if no inner tuples were found  {
16           If there are no valid outer tuples
                 and we have reached end of
                outer tuple then {
                Set tupStat to SUCCESS;
                return NULL;
             }
             else {
21              Set tupStat=LATER
                Poll AIO on the requests
                    enqueued in node's pending
                     requests queue ;
                if parent node can handle
                   LATER status
                     return NULL;
                else
26                   goto perform_join;
             }
        }
        Return result to the parent operator,
            one tuple per call
}
```

Figure 4: Asynchronous INL Iterator : Second stage

buffer as valid; the buffer will be used in subsequent iterations. The LATER status can potentially propagate upto the root.

6 Asynchronous Versions of Other Operators

In this section, we briefly discuss the asynchronous iterator support for various other operators.

- **Sequential Scan**: Sequential scan nodes are typically at the lowest level in the query plan tree. The sequential scan operator fetches a block from disk (or memory) and returns tuples one by one. When all the tuples from the current block are returned, a request for a new block is issued and the operator waits till it gets the block from disk. And the process of returning tuples starts.

  The simplest asynchronous version of sequential

```
  Tuple SeqScan(SeqScanState *seqscanstate,
      Status *retstat){

   Check for next tuple with buffer manager
5
   if(hit){ // Found tuple in main memory
     *retstat = SUCCESS;
     return tuple;
   }
10 else{
     Initiate an asynchronous fetch request to
         buffer manager
     *retstat = LATER;
     return NULL;
   }
15 }
```

Figure 5: Prototype for Asynchronous Sequential Scan without Out-of-Order Fetch

  scan does not block when a disk read is issued, and instead returns LATER, allowing other parts of the plan to proceed. We give the prototype for asynchronous sequential scan without out-of-order fetch in Figure 5.

  In many cases, the order of tuples has no importance. Hence, scanning the blocks strictly sequentially is not mandatory. A sequential scan fetches blocks in the order in which they are stored. With the increasing size of main memory, and correspondingly database buffer sizes, it is quite likely that many disk blocks that are required for a sequential scan are already present in buffer when the scan is initiated. A normal scan would block on the first access to a disk block that is not in the buffer. The main idea behind an "out of order" sequential scan is to look ahead in the scan, and if any blocks that would be accessed later are in the database buffer, tuples from those blocks can be returned ahead of tuples from blocks that are not currently in the buffer. Asynchronous IO operations are issued for blocks that are not in the buffer.

  We note that operating systems as well as database systems perform prefetching for sequential IO, hence asynchronous fetching can be expected to play a smaller role than out of order processing. We do not present numbers here, but some preliminary results showed that the gains were insignificant in our context of disk IO. However, asynchronous sequential scan may be useful for queries which can stop early (e.g. subqueries within an exists clause), and in other contexts such as distributed/web queries where some results may be in cache.

- **Sort**: Sort is an example of an operation that does not permit out of order processing, and it may appear that the asynchronous iterator model

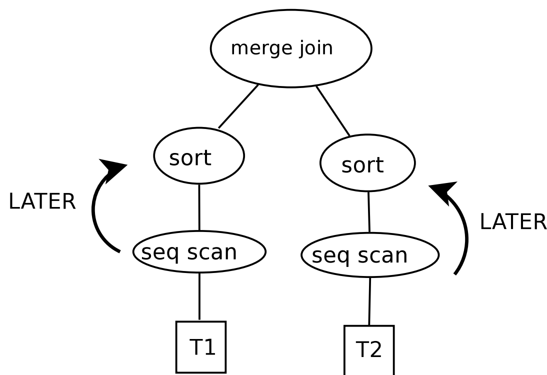

Figure 6: Asynchronous Iteration with Sort/Merge

is not very useful in this case. However, consider the example of a sort node with a sequential scan operator as a child. If the sequential scan node returns a LATER status, the sort node can return a LATER status to its parent node, and can be reinvoked after some time to continue sorting. Suppose this sort node is the left child of a merge join as shown in Figure 6. If a merge join node gets a LATER status from its left child, it could decide to start or continue processing the right child, and vice versa, instead of blocking. In this way, we get a higher utilization of both CPU and the disk, even though out-of-order processing is not possible.

- **Index Lookup**: Asynchronous IO can be applied for index lookups as well. The synchronous IO times involved in processing large indexes can be significant. Asynchronous IO can be used to fetch index blocks into the memory and the index lookup can return a LATER status if a block is not in the memory after issuing an AIO read.

7 Implementing Asynchronous Operations in PostgreSQL

In this section, we discuss how we modified the PostgreSQL database to support asynchronous versions of operations like file read and buffer management. The codebase we used was PostgreSQL 8.1.3.

7.1 File Read

The normal FileRead operation of PostgreSQL places a read function call to read in the file contents. The call read(file_id,buffer,amount) is a blocking call and the process waits for it to complete. We change this read call to an asynchronous version and allow the process to continue its processing. We simply replace the blocking function as shown in Figure 7.

We issue a LIST AIO request at the end of each iteration of fetch_outerbatch of Async INL, for all the new requests in the nodes AIO array.

```
aioreq = node's AIO array
aioreq->aio_fildes = file descriptor;
aioreq->aio_nbytes = amount;
aioreq->aio_buf = buffer;
aioreq->aio_offset = offset;
aioreq->aio_lio_opcode = LIO_READ;
Enqueue request in node's AIO array ;
EnQueue request in node's pending_req queue;
```

Figure 7: Asynchronous File Read

```
retVal = lio_listio(LIO_NOWAIT,
       node->aioarray, total_Reqs, no handler);
```

7.2 Buffer Read

ReadBuffer returns a pinned buffer containing the requested block of the requested relation. The ReadBuffer function first checks if the requested buffer is in the memory. If its found in the memory, only some statistics need to be updated.

The ReadBuffer uses storage manager routines to invoke FileRead function which is a blocking function. Since, we have changed this to a non-blocking call and return early, we split the ReadBuffer function into two parts, ReadBufferInit which calls this file read function and ReadBufferFinish which updates the flags and statistics when the read has completed. The corresponding functions at the storage manager level are also made asynchronous by splitting them into two functions, one which does the initialization and one which completes the book-keeping once the asynchronous operation is complete.

7.3 Polling for AIO Completion

Whenever we have no tuple to join in an INL node, we poll for AIO completion. We pass the node's pending request array as an argument. We use aio_error function to check the status of IO. This function returns a 0 on IO completion. If an IO request is completed, we make the associated buffer valid by calling the ReadBufferFinish function. Also, we dequeue that IO request from node's pending queue. From our experiments, we observe that the polling model performs better than the interrupt-based model. One such query run in shown in Figure 8.[1] In this graph, as well as in other graphs in the performance study, we show the time taken to output the ith tuple. As can be seen from the figure, polling requires about 10% less time compared to interrupts, at different numbers of tuples fetched. We use the polling model for all our experiments.

[1] myorders is the TPC-H orders relation stored sorted on o_custkey

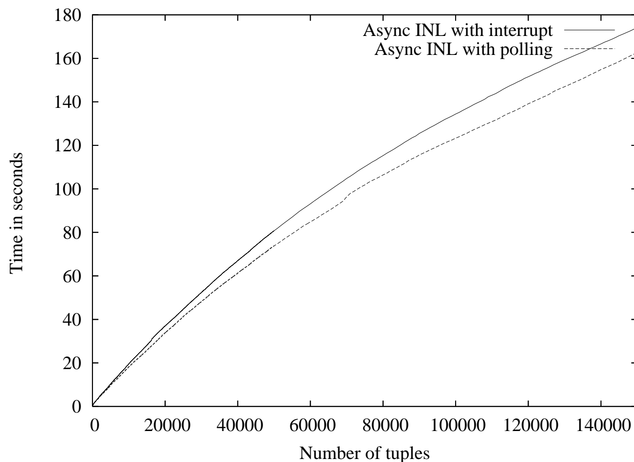

Figure 8: ***select l_orderkey from myorders, lineitem where o_orderkey=l_orderkey ;***

7.4 Start-up Effect

Since we operate on a batch of outer and inner tuples in each iteration, Async INL looses time when compared to (original) blocking INL for some initial number of tuples, after which it starts winning. If the initial batch size is very high, Async INL will take a much longer time than INL to generate the first few tuples. To mitigate this problem, we increase the BATCH_SIZE of inner tuples in a geometric manner. In our implementation, we start from a batch size of 1 and multiply it by a factor of 2 each time till we reach a maximum limit. The maximum limit is restricted by the number of concurrent asynchronous IOs that the system can support. Further, as we increase the number of concurrent requests, the system overheads appear to increase. Experimentally, we found that a maximum batch size of 200 performs best.

7.5 Other Issues

An interesting issue we faced was that asynchronous IO implementation uses threads, which violated an assumption about stack contiguity made by the check_stack_depth function, which is used within PostgreSQL code to detect uncontrolled recursion. (Checking for uncontrolled recursion is required since PostgreSQL supports SQL functions, which can be recursively defined, and are evaluated by recursively defined code in PostgreSQL.) This function erroneously indicated excessive recursion even in the absence of recursion, and had to be disabled as a result. A thread-safe implementation of this function is an area of future work.

Currently asynchronous versions of INL and sequential scan (with potentially out-of-order delivery of tuples) are used where ever the original plan had a synchronous version of the operator. We need to check the parent/ancestor operators to see if they depend on any ordering provided by the synchronous version of the operator, before using the asynchronous version of the operator with out-of-order delivery. Modifying the optimizer to choose the best plan taking asynchronous versions of operators into account is another area of future work.

8 Experimental Evaluation

In this section, we give a detailed performance study of asynchronous iterator for INL joins. We have performed our experiments with TPC-H database with scale factors of 1 and 10 in three different setups listed below.

1. Core 2 duo P4 with 1GB RAM and TPC-H - 1 GB database (single disk)
2. Core 2 duo P4 with 1GB RAM and TPC-H - 10 GB database (single disk)
3. Core 2 duo P4 with 3.2GB RAM and TPC-H - 10 GB database (4 disks / RAID 10)

Our experiments used PostgreSQL 8.1.3 as the code base, and compared it with our modified version of the same code base.

To ensure consistent results, before each query evaluation, we force the Linux kernel to drop the page cache, inode and dentry caches. On Linux kernels from version 2.6.16 upwards, this can be done by executing the command `echo 3 > /proc/sys/vm/drop_caches`, after executing the `sync` command to flush dirty pages back to disk.

We give the performance benefits of Async INL as compared to original INL, as well as compared to merge join, for several different join queries. All graphs have the number of tuples output as the X-axis, and the time taken as Y-axis; a point on the graph at $X = i$ shows the time taken to output the ith tuple.

For some queries, INL is the default plan chosen by the optimizer. For such queries, we provide timing numbers up to large enough number of tuples to meaningfully compare the plans. For other queries, we provide results up to the point at which merge join becomes cheaper than Async INL; this is the meaningful range for comparing Async INL with INL. For the query runs, we increase the size of shared buffer of PostgreSQL to 3900 (limited by the OS kernel) and the size of work memory to 256MB given that main memory is 1GB or more.

8.1 Experiments with TPC-H 1GB database and 1 GB RAM

In this section, we present the queries performed with TPC-H 1GB database and 1GB RAM.

We first consider a simple join of two relations, orders and lineitem, with a selection condition on l_orderkey which selects 1 in 100 tuples from lineitem.

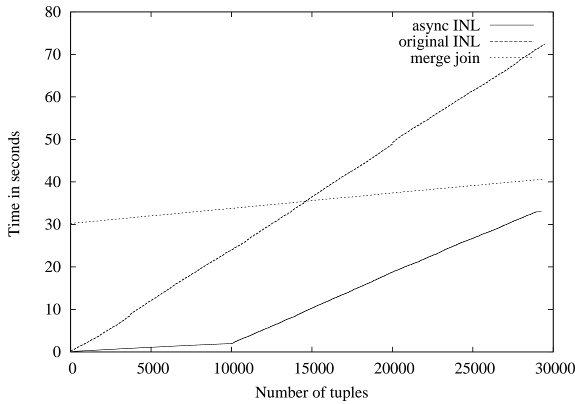

Figure 9: Query 1a [TPC-H 1 GB / 1 GB RAM]

Query 1a: select l_orderkey,l_quantity from orders,lineitem where o_orderkey=l_orderkey and l_orderkey%100=2 and l_linestatus='F'

Without the selection condition, merge join is the clear choice, since both relations are clustered on the join column. With the selection, only a small fraction of lineitem tuples need to be fetched, and the PostgreSQL optimizer chooses INL join for this query. As we can see from Figure 9, which shows results to completion of the query, there is a gain of *54%* after 29200 tuples for Async INL over INL and a gain of about *18%* over merge join. The gain over merge join is much higher at a smaller number of tuples.

The next query dropped the selection condition, but replaced the orders relation in the above query with an identical relation myorders, whose only difference is that it is not sorted on o_orderkey.

Query 1b: select l_orderkey,l_quantity from myorders, lineitem where o_orderkey=l_orderkey

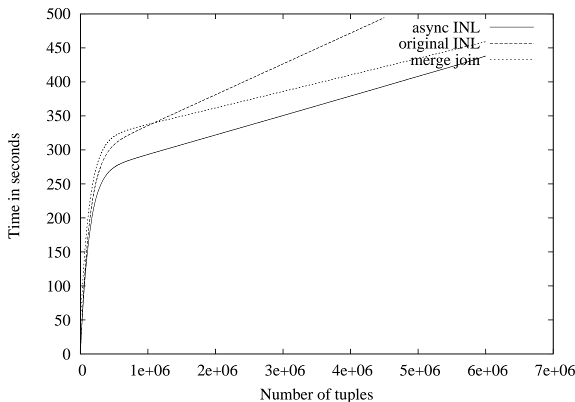

Figure 10: Query 1b [TPC-H 1 GB / 1 GB RAM]

As we can see from Figure 10, Async INL shows a gain of over *10%* compared to INL after 1,000,000 tuples, while the benefit over merge join is nearly 20% at this point. Since myorders relation is not clustered on orderkey, each access to it involves a random IO. Therefore the join of myorders and lineitem gives a gain even without a selection condition. We do not present results for the join of orders with lineitem, since neither version of INL is a good choice for this case even for a small number of tuples output.

The next query uses a three-way join of orders, lineitem and customer, with a filter that selects 1 in 100 orders.

Query 2a: select l_orderkey,l_quantity from orders,lineitem,customer where o_orderkey=l_orderkey and o_custkey=c_custkey and l_orderkey%100=2 and l_linestatus='F'

As we can see from Figure 11, for this query Async

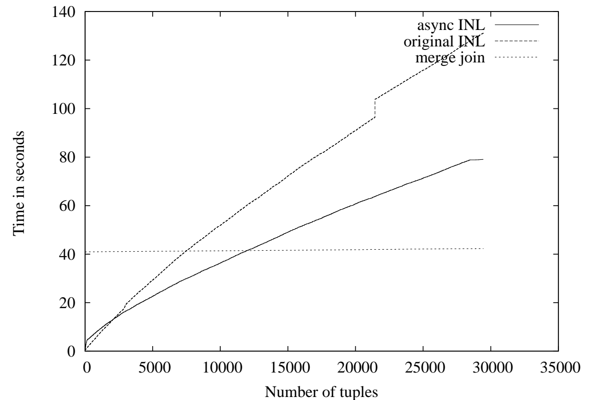

Figure 11: Query 2a [TPC-H 1 GB / 1 GB RAM]

INL shows a gain of over *25%* compared to INL and to merge join after 7000 tuples. Merge join beats Async INL after about 12000 tuples.

In the next query, we replace the orders relation with myorders and remove the filter.

Query 2b: select l_orderkey from myorders, lineitem, customer where o_orderkey=l_orderkey and o_custkey=c_custkey

As we can see from Figure 12, Async INL shows a gain of over *6%* compared to INL and to merge join after 100000 tuples.

We next consider Query 12 of the TPC-H benchmark.

TPC-H Q12: select l_shipmode,sum(...) from orders,lineitem where o_orderkey = l_orderkey and <several selection> group by l_shipmode order by l_shipmode

This query uses aggregation, and has only 2 results. On the PostgreSQL system, this query had INL in the chosen plan. (Only one other query had INL as the optimal plan, but it used subqueries which are not currently handled in an asynchronous fashion in our implementation, so we do not report results for that query.) For this query, the asynchronous INL completes in 48 seconds as against 64.7 sec for the original INL, giving a gain of over 25%.

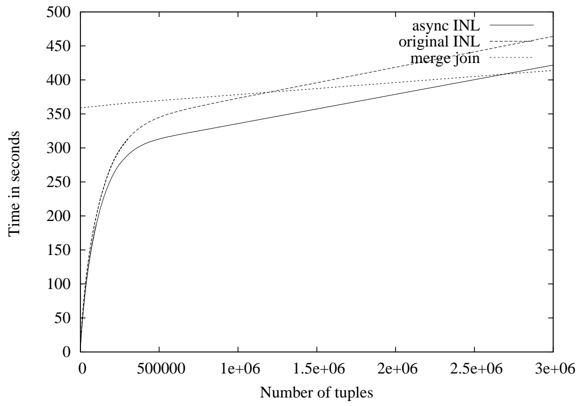

Figure 12: Query 2b [TPC-H 1 GB / 1 GB RAM]

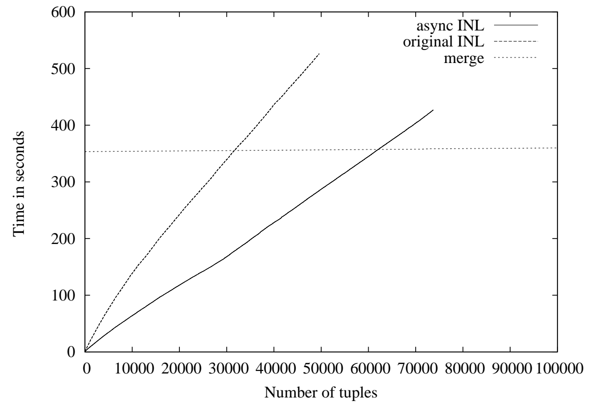

Figure 14: Query 2a [TPC-H 10 GB / 1 GB RAM]

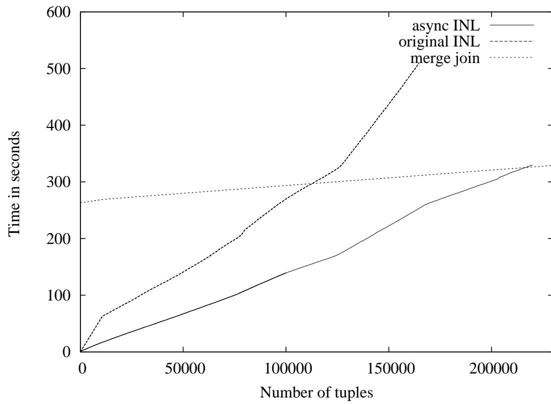

Figure 13: Query 1a [TPC-H 10 GB / 1 GB RAM]

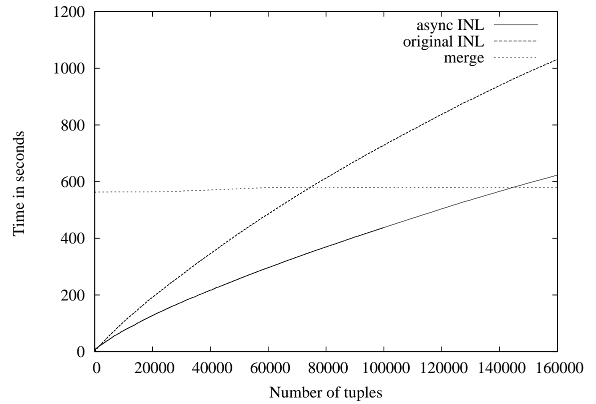

Figure 15: Query 2b [TPC-H 10 GB / 1 GB RAM]

8.2 Experiments with TPC-H 10GB database and 1 GB RAM

In this section, we present the queries performed with TPC-H 10GB database and 1GB RAM. Figure 13 shows the performance of Query 1a (which we saw earlier) on the TPC-H 1 GB database, with 1 GB of RAM. As we can see from the figure Async INL shows a gain of nearly *50%* over INL, and over *50%* compared to merge join, at 1,00,000 tuples.

Figure 14 shows the performance of Query 2a. As we can see from the figure, Async INL shows a gain of over *50%* compared to both INL and merge join at 30,000 tuples. For Query 2b, Figure 15 shows that Async INL gives a gain of nearly *40%* as compared to INL and merge join after 65,000 tuples.

For TPC-H Q12 presented earlier, the asynchronous iterator completes in 431 sec as against 687 sec for the original giving a gain of 37%.

Comparing the results on the TPC-H 1GB database with those on the TPC-H 10GB database, we see a few patterns emerge. First, the number of tuples at which merge join becomes cheaper than INL or Async INL is significantly larger in the latter case, since much larger relations need to be sorted for merge join. Second, the time taken to output the ith tuple increases significantly in the latter case. Third, the relative benefit of Async INL over INL also increases significantly in the latter case. We believe the latter two patterns are because less of each relation fits in the buffer, reducing the probability of a random access finding the required data in the buffer. As a result IO time would increase significantly, and Async IO correspondingly gives more benefits.

8.3 Experiments with TPC-H 10GB database and 4-disks RAID-10 with 3.2 GB RAM

For our next set of experiments, we use a 4-disk RAID 10 for our experiments. In this configuration, data is striped across two disks, which are mirrored on two other disks. IO operations can be executed on either of a mirrored pair of disks.

As we can see from Figure 16, for Query 1a we have a gain of nearly 20% after about 300,000 tuples with Async INL; in this case INL was chosen over merge join by the PostgreSQL optimizer. Merge join was uniformly worse than Async INL and INL up to completion of the query.

For Query 1b, as we can see from Figure 17, we have a gain of about 50% after 80,000 tuples with Async INL; after this point merge join becomes the best al-

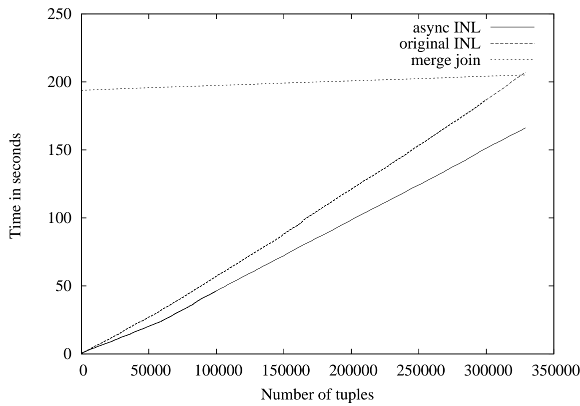

Figure 16: Query 1a [TPC-H 10GB / 3.2 GB RAM / 4-disks RAID10]

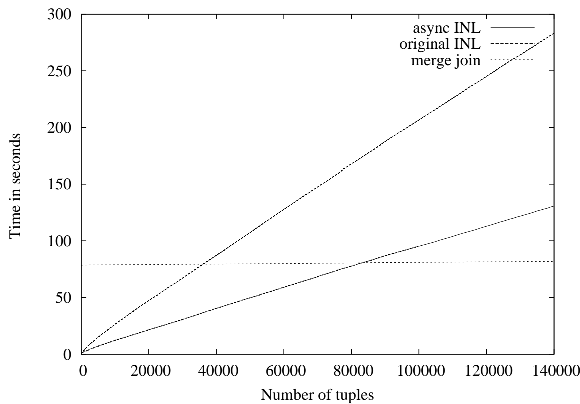

Figure 17: Query 1b [TPC-H 10GB / 3.2 GB RAM / 4-disks RAID10]

gorithm, but Async INL continues to beat INL by a similar margin.

For Query 2a, Async INL shows a gain of nearly 30% compared to both INL and merge join at 125,000 tuples as shown in Figure 18; as for Query 1a, INL was chosen over merge join by the PostgreSQL optimizer.

For Query 2b, as we can see from Figure 19, Async INL shows a gain of nearly 55% compared to INL and to merge join at 20,000 tuples.

For TPC-H Q12, presented earlier, the asynchronous iterator completes in 147.6 sec as against 164.06 sec for the original giving a gain of 10%.

Query 4: select c_name,l_linenumber,l_partkey, l_shipmode,s_suppkey,s_name from orders, lineitem, customer, supplier where o_orderkey=l_orderkey and o_custkey=c_custkey and l_suppkey=s_suppkey;

As we can see from Figure 20 , for this query the asynchronous INL plan gets a gain of *27 %* after 90,000 tuples compared to the original INL plan. Since the selection clause has c_name and s_name, an index-only plan is not applicable here. The merge join plan for this query required a sort of very large relations, and took nearly 50 minutes to generate even the first tuple; the 90,000th tuple also took about the same time.

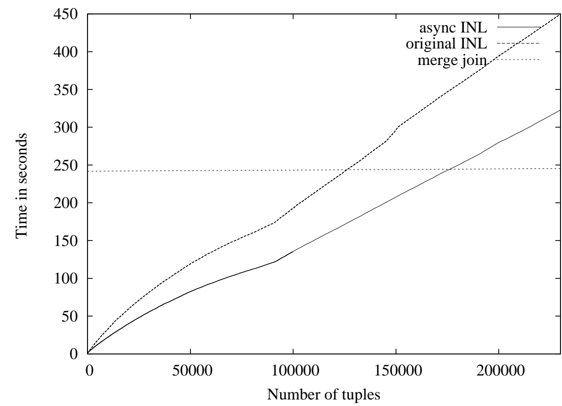

Figure 18: Query 2a [TPC-H 10GB / 3.2 MB RAM/ 4-disks RAID 10]

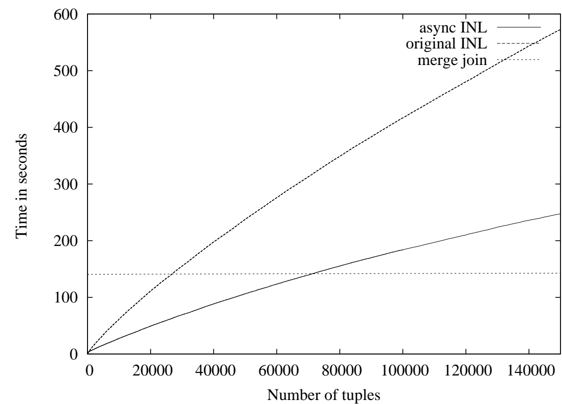

Figure 19: Query 2b [TPC-H 10GB / 3.2 MB RAM/ 4-disks RAID 10]

Comparing the results for queries on the configuration with 1 GB of RAM and 1 disk, to the configuration with 3.2 GB of RAM and RAID 10 with 4 disks, we observe several interesting phenomena. First, across

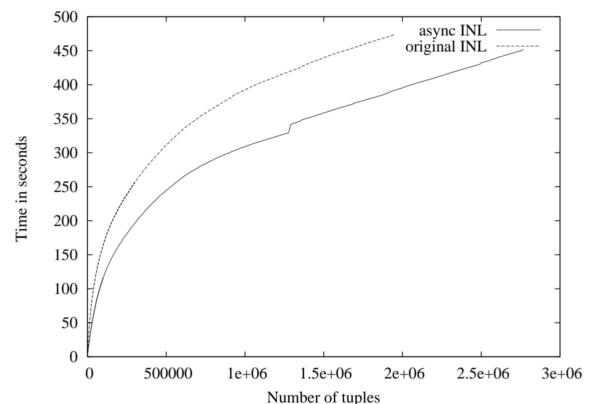

Figure 20: Query 4 [TPC-H 10GB / 3.2GB RAM / 4-disks RAID 10]

all queries the time taken for INL, Async INL and merge join all decrease considerably with the second configuration. Second, the decrease in time taken is much more for INL and Async INL, compared with merge join. As a result, the number of tuples at which merge join beats INL or Async INL increases sharply in the second configuration. Third, the relative benefit of Async INL as compared to INL in the two configuration is not uniform across queries. Async INL always beats INL by a significant margin, but the margin increases on the second configuration for some queries, while it decreases for others. We conjecture that this effect may be due to the memory size being much larger, resulting in many random IO operations finding required data already in memory due to OS level prefetching. Thus, even though a system with multiple disks can be expected to benefit more from Async INL, the observed relative benefit due to Async INL is less in this configuration.

9 Conclusion and Future Work

Asynchronous IO overlaps computation and IO processing of multiple IO requests and avoids the blocking problem of regular IO. We propose an Asynchronous Iterator Model which exploits asynchronous IO and avoids the blocking problem. In this framework, a node need not always return a tuple, it can return a LATER status instead and we can proceed with the CPU processing which does not depend on that IO request. We also present an Asynchronous Index Nested Loop join based on this framework. We have evaluated this model with number of queries based on the TPC-H benchmark database, and demonstrated gains of over *50 %* for several queries in a (4 disk) RAID 10 setup.

There are several further opportunities for exploiting asynchronous IO within a database system such as PostgreSQL. For example, B-tree access is currently blocking, although fetching of records is asynchronous. Modifying B-tree code to support asynchronous IO is an important task. Nested subplans currently result in blocking of the parent operator. Supporting asynchronous iteration at this level would be another area of future work. Implementing asynchronous versions of other operations, such as sort and merge join, is also part of future work.

At a broader level, the asynchronous iterator model can be useful in other settings. For example, systems that consume web services in bulk can use the Asynchronous Iterator Model to hide the latency inherent in executing operations at a remote site, allowing much higher throughput rates when accessing web services. Similarly, data integration systems such as IBM Data Joiner can benefit from asynchronous IO when fetching data from remote sites.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *In SIGMOD*, pages 261–272, 2000.

[2] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *SIGMOD*, pages 285–296, 2000.

[3] G. Graefe. Query evaluation techniques for large databases. In *ACM SIGMOD*, Washington, D.C., May 1993.

[4] G. Graefe. Executing nested queries. In *Database Systems for Business, Technology and the Web*, University of Leipzig, Germany, Feb 2003.

[5] http://www.ixora.com.au/notes/asynchronous_io.htm .

[6] http://www.oracle-base.com/articles/misc/DirectAndAsynchronousIO.php .

[7] http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/core/c0011638.htm .

[8] V. Raman, V. Raman, and A. Deshpande. Using state modules for adaptive query processing. In *ICDE*, pages 353–364, 2003.