ALACRITY Release Document

April 22, 2016

1 ALACRITY Introduction

1.1 What is ALACRITY ?

ALACRITY is a lightweight query engine based on PForDelta compressed inverted indexes over scientific data. Its purpose is to accelerate response times on exploratory query-driven analysis and visualization. It is designed to function either during post processing or in situ (while a simulation application is running concurrently). Its core idea is to binning data based on significant bits in the IEEE floating-point representation and thus, the search space is reduced to a subset of bins by excluding bins that definitely are not in the querying condition.

The below sections give the details of ALACRITY, in which Section **??** presents how ALACRITY does the binning (referred as indexing) while Section **??** describes how ALACRITY answers queries based on indexed built previously.

1.1.1 Indexing

ALACRITY's speciality is for scientific data, which is floating-point values. The key observation is that, while floating-point datasets have a large number of unique values, the highest k bytes (significant bits) of the datasets typically have lower cardinality (number of unique values) compared to the rest of the data. Thus, its core strategy is to binning on the high-order bytes/significant bits of the data, while storing the low-order bytes separately, and subsequently applying compression methods on resultant Inverted Indexes.

Figure **??** demonstrates the significant-bit-based binning strategy (also referred as *encoding*) employed in ALACRITY. The input data is treated as a linear data array, and each data is implicitly associated with an Row ID (RID) starting with 0. Every data is examined under the two-part split, while the high-order bytes (significant bits) produce bins, in which RIDs of the same-bin data are organized into one bin (referred as inverted indexes), and low-order bytes bin are correspondingly concatenated as a low-order-

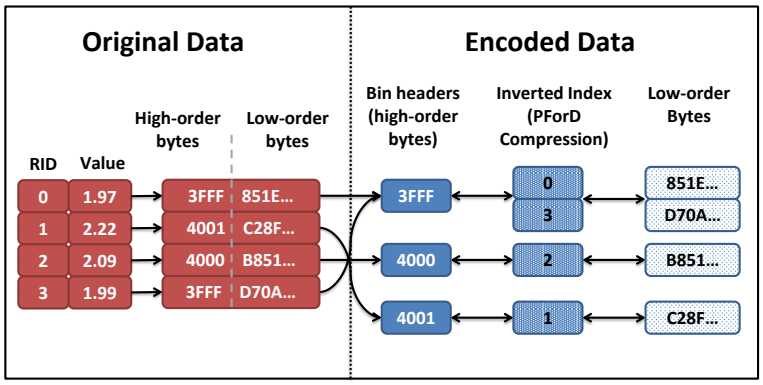

Figure 1: An overview of how raw floating-point data is encoded by ALACRITY for compact storage and query-optimized access.

byte bin. These inverted indexes are further compressed by the PForDelta compression method to reduce the storage footage.

The implementation of the encoding method is the function *ALEncode* defined in *src/ALEncode.c*. It has two scans. The first scan builds the bin layout (see the function *determineBinLayout*) by counting the number of elements in each bin (all possible bins are enumerated at first). The second scan splits every data, and put its associated RID into the bin and its lower-order bytes data in the corresponding lower-order byte bin (see the function *encodeWithInvertedIndex*).

The command of build ALACRITY indexes Building ALACRITY is relatively easy as we only need one single command. The command looks like this:

${ENCODE} encode -p${PSIZE}E -x -e${DATA_TYPE} -s${SIG_BITS} ${PATH_TO_ORIG_DATA} ${PATH_TO_OUTPUT}

where ${ENCODE} is the path to the executable *alac*, ${PSIZE} is the number of elements in each partition, ${DATA_TYPE} could be *double* or *single* indicates whether the data is double or single floating point data, ${SIG_BITS} is the number of significant bits used in the binning, ${PATH_TO_ORIG_DATA} is the path to the original data which is a binary binary (unfortunately, that is the only input data format ALACRITY now supports), and the ${PATH_TO_OUTPUT} is the path to the index output. Additionally, *-x* indicates inverted indexes are compressed by the PForDelta compression. If you want inverted indexes without any compression, please replace *-x* with *-i*.

Notice that there is an partition option in the encoding command, this is because ALACRITY divides entire original input data into partitions to prevent the overflow. It works well for extreme large datasets. Yet, it also has an shortcoming, that is the number element of one partition can not

exceed 2^{32} since the internal data type of RID is *uint32_t*. In addition, there are other ways of specific the partition size, I do not cover them for the sake of space. Please refer to the *alacrity.c* in *tools* folder for details.

Blow is an example of encoding command. It means encoding the original double precision data using 16 significant bits with PForDelta compression, and each partition has 20 millions elements. One more thing I would like to point out is that the ${PATH_TO_OUTPUT} has the prefix (in this case, it is *temp*), and temp-compressed_data.dat (low-order byte data), temp-metadata.dat (ALACRITY's metadata), and temp-query_index.dat (inverted index either compressed or non-compressed) are generated.

build/bin/alac encode -p20000000E -x -edouble -s16 /home/chris/temp.bin /home/chris/temp

1.1.2 Querying

ALACRITY's querying ability supports RID-retrieval queries with multi-variate constraints (e.g., select *RIDs* from *ALACRITY-format indexes* where $5 < temperature < 18$ and $20 \leq pressure < 60$) . Differ from the query APIs designed in ADIOS (see Section **??** for details), it does not support a spatial constraint. To understand the ALACRITY query processing, we first present the query processing for uni-variate query (e.g., select *RIDs* from *ALACRITY-format indexes* where $5 < temperature < 18$), which is explained step by step below with the flowchart (Figure **??**).

1. ALACRITY metadata load Loading ALACRITY's metadata is an beginning step and it mainly reads the bin layout information created during the encoding step. Please see the details in the function *getPartitionMetadata*.

2. Bin "touching" determination

   Use the input value query constraint to determine how many bins falls in the value constraint (see function *findBinRange1C*). If there is no bin touching, program exists; otherwise, proceeds to next step.

3. "Touched" bins' indexes load

   Read indexes (either compressed or uncompressed) for every "touched" bin (see the function *ALPartitionStoreReadIndexBins*)

4. Filter RIDs from "touched" bins

   At this step, it outputs resultant RIDs based on "touched" bins. There are two categories bins among in "touched" bins. One type of bin is edge bins, which are top and bottom bins in the "touched" bin list and require to be further examined (refereed as *candidate check*). The reason is that some RIDs might not in the resultant RID list. And the

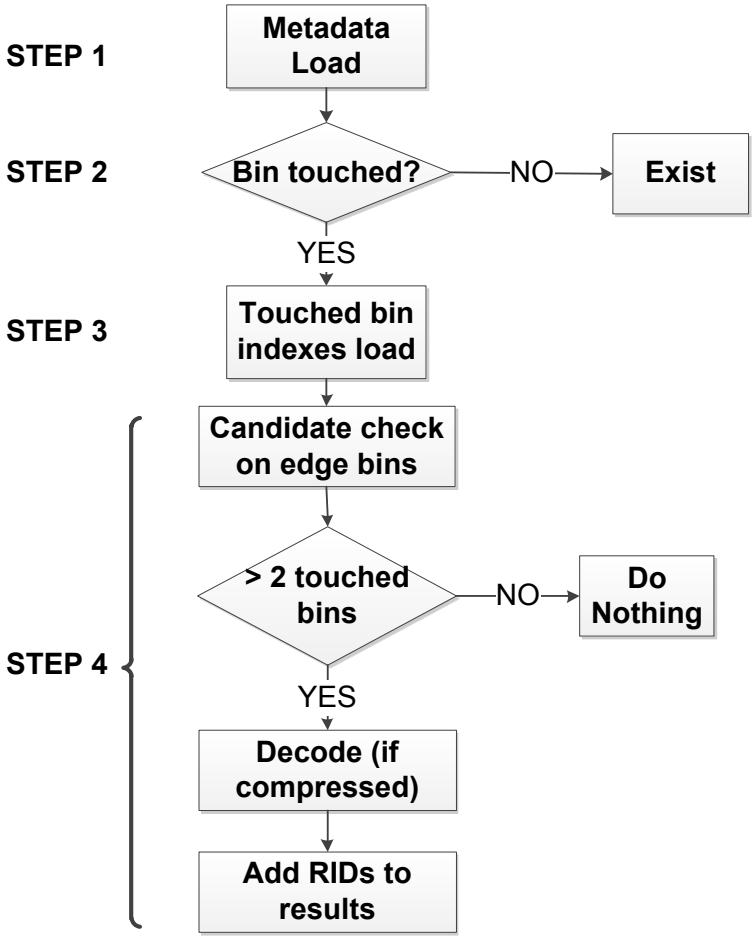

Figure 2: A flowchart of ALACRITY's query processing for uni-variate queries

criteria of the determination depends on whether the corresponding value satisfies the query constraint or not. Another type of bin is internal bins, which are bins except edge bins, and do not need the further examination, meaning all RIDs in these internal bins go to the results. Additionally, internal bins only exist if only the number of "touched" bins is more than 2. For example, if bin 3, 4, 5 and 6 are "touched" bins, bin 3 and 6 are *edge* bins need to be examined, and bin 4 and 5 are internal bins. Note that if bins are in compressed format, the decompression occurs before. In addition, in the *candidate check* step (see the function *candidateCheck*), the original data is first recovered first by assemble the high-order and low-order bytes (see the function *readAndReconstituteData*) and the query constraint is applied

on the recovered data during the scan.

Now, we move on to the query processing for multi-variate queries, in which a temporal result obtained from processing one query constraint is represented by a bitmap. These bitmaps are performed by bitwise operations (depends on the keywords, i.e., AND, OR, and NOT) after all query constraints are processed. Note that processing one query constraint is almost same as the univariate query processing, except in the last step, it outputs a bitmap, rather than RIDs, which requires an additional step to sets RID in the bitmap. Figure **??** shows overview of the multi-variate query processing, please check the *tools/multiquery.c* for details. As mentioned, the step 1 in the Figure **??** is replaced with uni-variate query processing.

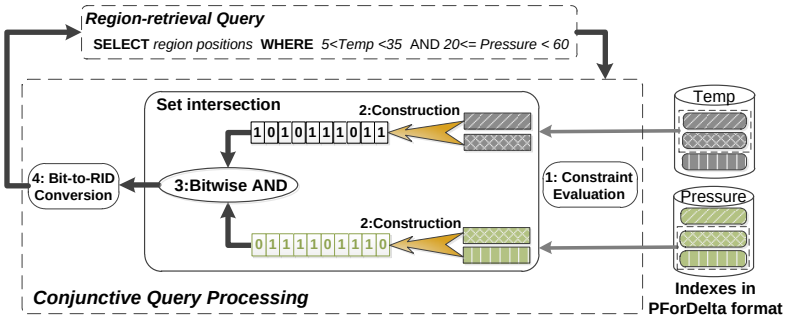

Figure 3: An overview of multi-variate query processing, which is composite of multiple independent univariate query processing through bitmaps.

2 ALACRITY within ADIOS Introduction

2.1 How does ALACRITY within ADIOS differ from ALACRITY alone?

There are three main differences between the pure ALACRITY and ALACRITY within ADIOS, they are:

1. The query APIs designed in ADIOS is more complicated than the query interface in pure ALACRITY.Specifically, the query APIs not only have value constraints, but also a spatial constraint (denoted as input bounding box B). Note that the spatial constraint could be a *ADIOS_SELECTION_WRITEBLOCK*, but for illustration purpose, I only use the *ADIOS_SELECTION_BOUNDINGBOX* since the process of *ADIOS_SELECTION_WRITEBLOCK* is similar to that of *ADIOS_SELECTION_BOUNDINGBOX*. In addition, the query APIs

returns coordinates to users, in which the returning coordinates are relative to input bounding box B and they are retrieved in multi-batch manner.

2. The index building of ALACRITY within ADIOS is accomplished through ADIOS transformer layer, which means the index building is achieved in *in situ* fashion (while the data is writing out by the application, the index is built at the same time), as oppose to a post-processing manner in the pure ALACRITYThis also means the ALACRITY indexes are stored in BP format in contrast to binary file format in pure ALACRITYTo the developer, the discrepancy of index format causes different APIs of reading data (especially meta, indexes, and low-order bytes data) from ALACRITY.

3. The differences of query processing in ALACRITY within ADIOS are primarily caused from the ADIOS query APIs, besides the different APIs of reading ALACRITY data. In particularly, the query processing in ALACRITY within ADIOS has to deal with the spatial constraint, ensuring the resultant coordinates to being in the input bounding box B.

Sections below document details of each point. Unless otherwise stated, ALACRITY below refers to the ALACRITY within ADIOS. In addition, all source codes mentioned below refer to codes in *src/query/query_alac.c*, except source codes in section **??** refer to codes in *src/transforms/adios_transform_alacrity_write.c*.

2.1.1 APIs

There are two main aspects ALACRITY responds to ADIOS query APIs worth to detail.

1. query-compose APIs

   As ADIOS query APIs allow end-users to compose queries with different value constraints. The ALACRITY internally has a tree structure to represent each individual query constraint and the composition relation between individual query constraints relying on *ADIOS_QUERY* structure defined in *adios_query.h*. The tree structure is processed in bottom-up fashion recursively, and each individual query is processed independently (please refer to the function *adios_alac_process*)

2. the result-retrieval API

   End-users can retrieve resultant coordinates computed by ALACRITY in batch manner, which means a desired number of results is specified by end-users. To achieve this functionality, the ALACRITY internally represents final results with a uncompressed bitmap structure (basically, it is an array of *uint64_t*. please see the *ADIOS_ALAC_BITMAP*

structure. The reason for using the bitmap is that it has fast speed of processing query keywords (i.e., AND, OR and NOT), which is achieved by corresponding bitwise operations. In addition to the bitmap itself, we track the position of results so far are retrieved by users. This position is indicated by the field (*lastConvRid*) in the *ADIOS_ALAC_BITMAP* structure. Note that ALACRITY does not keep the status of the *ADIOS_ALAC_BITMAP* structure, instead we let the user to keep it. This information is kept in the field *queryInternal* of *ADIOS_QUERY* structure. The conversion and de-conversion between them are performed by functions *convertALACBitmapTomemstream* and (convertMemstreamToALACBitmap), respectively.

One last thing I would like to point out is the bitmap-to-coordinates conversion. There is a discrepancy between the internal represent structure (i.e., bitmap) and user-desired data structure (i.e., coordinates), so we need to convert the bitmap to the coordinates, which is achieved by the function *adios_query_alac_retrieval_pointsNd*. The basic idea of this function is that we first move the position from last retrieval, read next N (where N is the user-specified batch retrieval size) bits, and convert all 1 bits to the coordinates. The conversion is performed by keeping dividing the RID (the position of an one bit sits) by the highest dimension (dimensions that changes fastest) in the function (*ridToCoordinates*).

2.1.2 Index Building

As mentioned, the ALACRITY index is built through transformer layer, which is seen in *src/transforms/adios_transform_alacrity_write.c* The basic idea (shown in the function *adios_transform_alacrity_apply*) is that when the transformer layer receives data from upper layers, it first parse the XML to obtain the configuration of ALACRITY index building (i.e., the number of significant bits, compression method to compress the RIDs, and whether keeps the lower-order bytes or the original data) and it then invokes the index build function (*ALEncode*, provided by the pure ALACRITY) to build indexes.

List **??** shows an example of showing input configurations for building ALACRITY indexes. The key component in this sample xml file is the "*transform*" tag, whose value begins with *alacrity*, following by several key-value pairs. Table **??** has a list of keys associate with ALACRITY index build. The source code of parsing the configuration is shown in the function *parse_configuration*.

As a result of this index building, every PG within BP file contains ALACRITY metadata, indexes, lower-order-byte data (if enabled) and original data (if enabled).

KEY	VALUE	COMMENT
IndexForm	ALInvertedIndex ALCompressedInvertedIndex	Inverted index, no compression Inverted index, with PForDelta compression
sigBits	number less than 64	number of significant bits
loBytes	true false	keep low-order-bytes do not keep low-order-bytes
origData	true false	keep original data do not keep original data

Table 1: Configurable options of ALACRITY index build via transformer layer

Listing 1: Sample configurable XML for ALACRITY index building

```
<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="S3D" coordination-communicator="comm">
            <var name="lx" type="integer"/>
            <var name="ly" type="integer"/>
            <var name="lz" type="integer"/>
            <var name="nx" type="integer"/>
            <var name="ny" type="integer"/>
            <var name="nz" type="integer"/>
            <var name="gx" type="integer"/>
            <var name="gy" type="integer"/>
            <var name="gz" type="integer"/>
            <var name="size" type="integer"/>
            <var name="rank" type="integer"/>
            <global-bounds dimensions="gx,gy,gz" offsets="lx,
                ly,lz">
                <var name="/temp" gwrite="t" type="double"
                    dimensions="nx,ny,nz" transform="
                    alacrity:indexForm=
                    ALCompressedInvertedIndex,loBytes=false,
                    sigBits=16,origData=false"/>
                <var name="/pressure" gwrite="t" type="double
                    " dimensions="nx,ny,nz" transform="
                    alacrity:indexForm=
                    ALCompressedInvertedIndex,sigBits=24,
                    origData=false"/>
                <var name="/uvel" gwrite="t" type="double"
                    dimensions="nx,ny,nz" transform="
                    alacrity:indexForm=
                    ALCompressedInvertedIndex,sigBits=16,
                    origData=true"/>

            </global-bounds>
        </adios-group>

<method group="S3D" method="MPI"/>

```

```
25 <buffer size-MB="6144" allocate-time="now" />
26
27 </adios-config>
```

2.1.3 Query Processing

As mentioned earlier, every individual query constraint is processed independently. Therefore, we only present processing steps for one query constraint, which is seen in the function *adios_alac_uniengine*. In addition, the process for the *ADIOS_SELECTION_BOUNDINGBOX* is similar to the *ADIOS_SELECTION_WRITEBLOCK*. I present the query process only for *ADIOS_SELECTION_BOUNDINGBOX* for simplicity. The idea is that for every query constraint processing, we first create an uncompressed bitmap, the presence of 1s represents the corresponding coordinates in the temporal results. We then compute the "touched" PGs (referred as candidate PGs) by invoking the function *adios_find_intersecting_pgs* provides from the transformer layer. In other words, the *adios_find_intersecting_pgs* function returns PGs are overlapping or fully contained in the input bounding box B, and discarding PGs not disjoint with B. Last, every PG in the candidate PG set are further examined to find exact results for the given query constraint. This is where the complexity comes from, and the implementation is seen at the function *proc_write_block*.

Now, I present the logic in the function *proc_write_block* step by step.

1. load ALACRITYplugin metadata by the function *read_alacrity_transform_metadata* provided by the transformer layer. As indexes are built within the transformer layer, it makes sense the query processing has to interactive with transformer layer to obtain any ALACRITY related information.

2. ALACRITYplugin metadata load

   Load ALACRITY metadata by the function *readPartitionMeta* to specific ALACRITY information, such as the bin layout and indexes.

3. Bin "touching" determination This is the same as the step in the Section **??**. If there is no bin, continue to process the next PG. Otherwise, proceeds to the next step.

4. PG-fully-contained determination

   Use ADIOS statistics information to determine whether the whole PG is fully contained in query constraints (both in spatial and value aspects). This step only applies when the resultant bins from the previous step span the entire bin.

5. "Touched" bins' indexes load

   This is the same as the step in the Section **??**. But this time, the indexes are read from transformed format, see the function *readIndexAmongBins* for details. In addition, the next step branches out depending on whether indexes are compressed or not. Yet, since the scenario of index compressed is more complicated than that of index uncompressed, I only present the index compression case.

6. Decode RIDs of a whole PG to the bitmap

   If a PG is fully contained, we can avoid the index decode step (i.e., decoding the compressed RIDs to RIDs) by taking advantages of the fact that is all RIDs need to be set. In other words, we rely on the prior knowledge that all RIDs must be RIDs from 0 to R, where $R =$ *PG_size* -1, which explains the reason why we do not need to decode the compressed RIDs.

   Additionally, we notice that RIDs that are on dimension that changes fastest are consecutive. We take this as another advantage, which allows us to set a consecutive 1 bits in the bitmap (see function *set_consecutive_bits*).

   Last, I would like to point out is that in the current implementation, this is implemented only for data up to 3 dimensions, which are the most common dimensionalities. Functions *setBits_for_wholePG_1_2_dim* and *setBits_for_wholePG_3_dim* are current implementation for 1 or 2 dimensions and 3 dimensions, respectively. The reason I have functions for different dimensions is that the stride between two consecutive bits depends on the dimensionality, making it hard to generalize. If dimensionality is higher than 3, we go back to original routine, (function *ALDecompressRIDtoSelBox*), which involves three tedious conversion (i.e., compressed RIDs to RIDs, RIDs to coordinates within B and coordinates to RIDs within B).

7. Set the bitmap for "touched" bins

   There are two categories bins among on "touched" bins. One type of bin is edge bins, which are top and bottom bins in the "touched" bin list and require to be further examined. Another type of bin is internal bins, which are bins except edge bins, and do not need the further examination. This is same as the pure ALACRITY.

   With regard to edge bins, we do a candidate check to determine which RIDs are in the result and which are not (see the function *adios_alac_check_candidate*). The major steps in the candidate check involves original data recovery and linear data scan (determining which value satisfies the query constraint or not), same as before. Note that if the value does satisfy the query constraint, it needs to be reflect in the

bitmap with its RID. And the bitmap set requires the RID-Coordinate-RID conversion (see the function *ridConversionWithoutCheck*). Furthermore, if current PG is partially overlaps with B, during the conversion, we need to do additional check, discarding those RIDs that are outside of B (see the function *ridConversionWithCheck*).

Whereas, for the internal bins, we set the bitmap by RIDs in bins. Here comes a second optimization, which aims for reduces the number of relatively expensive division and multiplication operations involved in the RID-Coordinate-RID conversion. This optimization is implemented by functions *deltaBitSetsWithPGPartialCoveredCorderIn3D* and *deltaBitSetsWithPGCoveredCorderIn3D*. The idea is to avoid the conversion for every RID by taking advantage of the current new RID based on the previous computed new RID is able to be calculated by addition operations. Again, because the stripe of consecutive RIDs depends on the dimensionality, we have to make case by case for each dimension.

3 Performance

3.1 Setup

We have performed all experiments on the Edison machine. And the test data we used is the S3D bp file with 200 PGs. In addition, all data is stored on GFS file system.

3.2 Indexing Performance

We have two performance metrics, index size and index building time in parallel.

3.3 Query Performance

3.3.1 Univariate Query

3.3.2 Multivariate Query

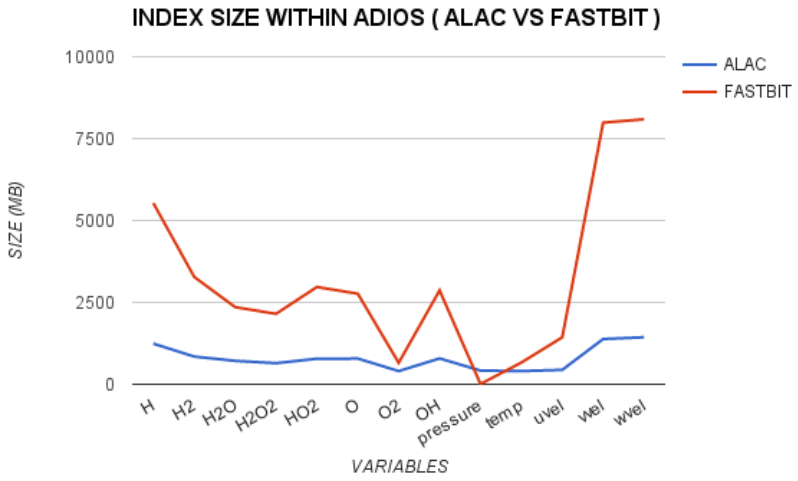

Figure 4: Index size comparison. In the ALACRITY index building, 24 significant bits are used on variable pressure and 16 significant bits are used on rest variables. Additionally, original data are discarded.

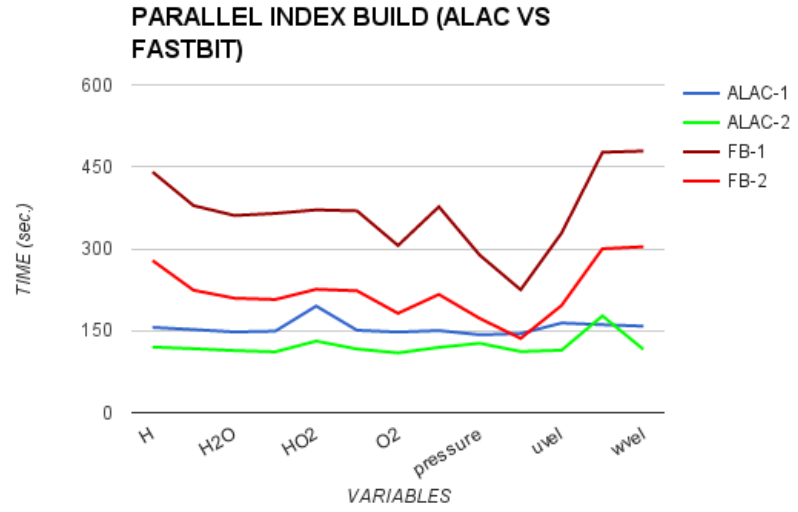

Figure 5: Index building speed comparison using 1 and 2 processes.

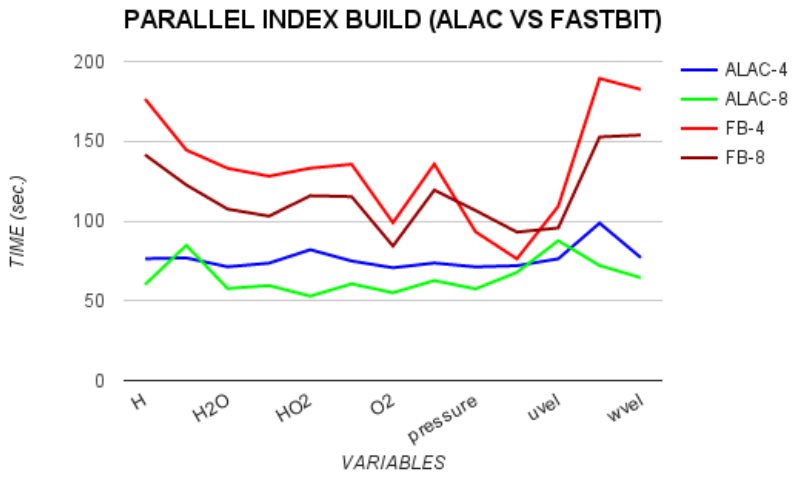

Figure 6: Index building speed comparison using 4 and 8 processes.

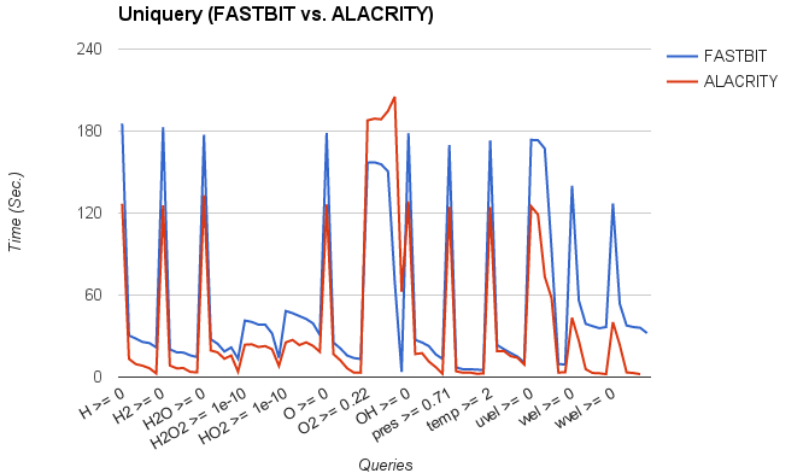

Figure 7: Performance comparison for univariate query with the *whole* spatial selection.

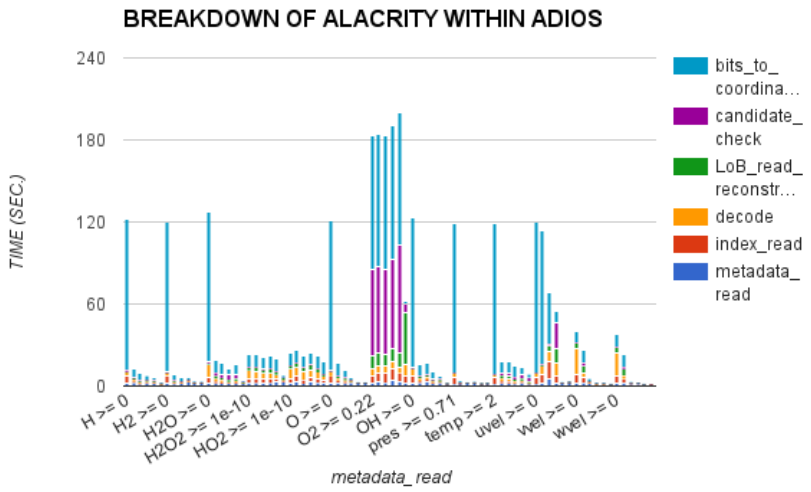

Figure 8: Breakdown performance comparison for univariate query with the *whole* spatial selection.

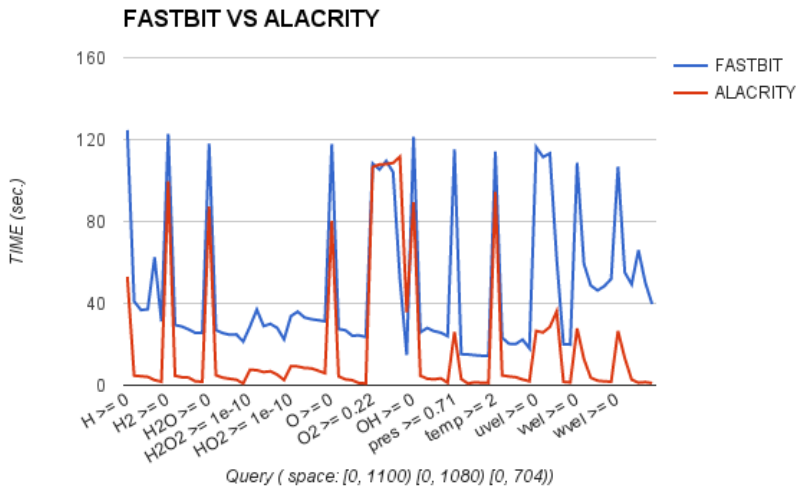

Figure 9: Performance comparison for univariate query with the *half* spatial selection.

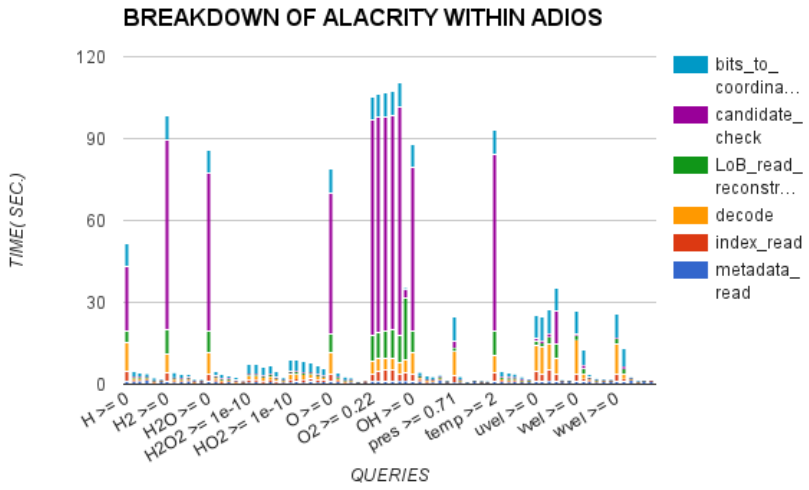

Figure 10: Breakdown performance comparison for univariate query with the *half* spatial selection.

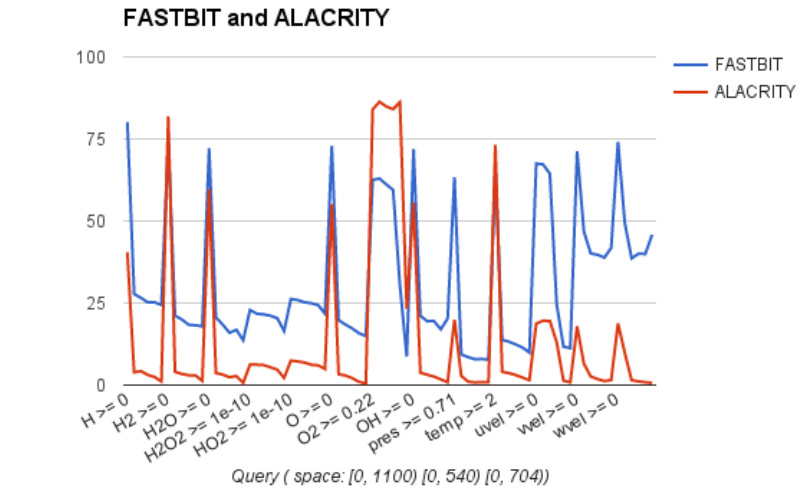

Figure 11: Performance comparison for univariate query with the *quarter* spatial selection.

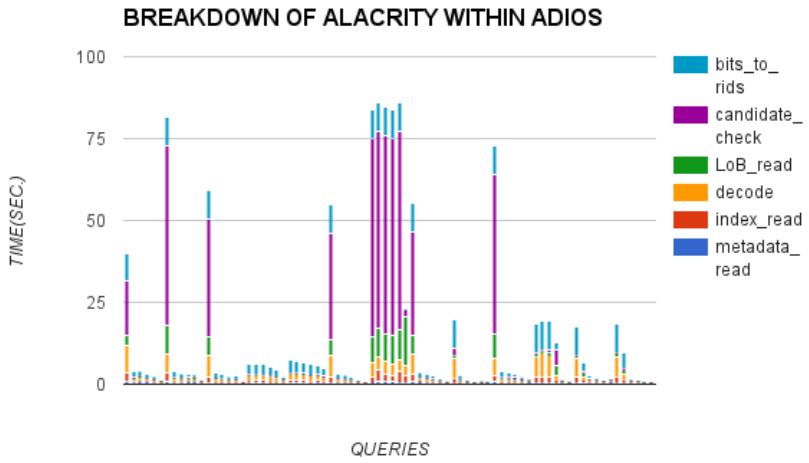

Figure 12: Breakdown performance comparison for univariate query with the *quarter* spatial selection.

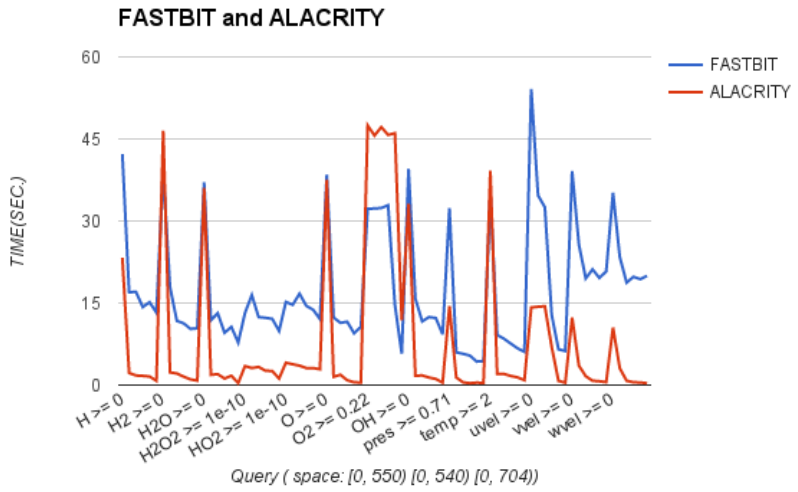

Figure 13: Performance comparison for univariate query with the *eighth* spatial selection.

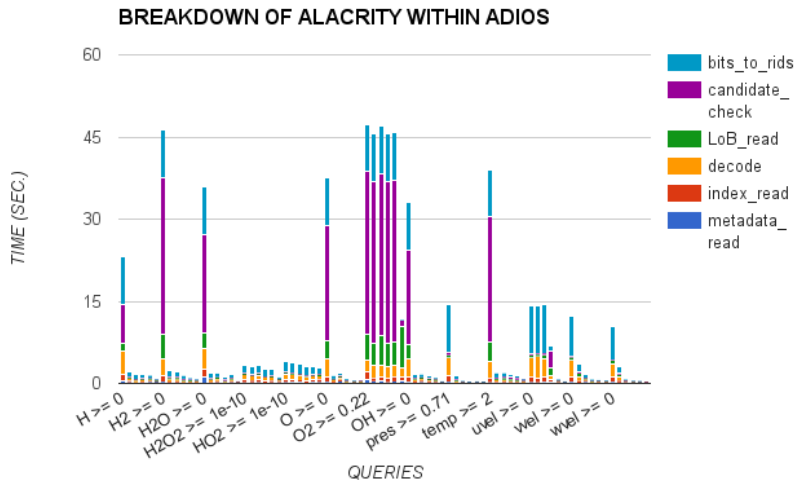

Figure 14: Breakdown performance comparison for univariate query with the *eighth* spatial selection.

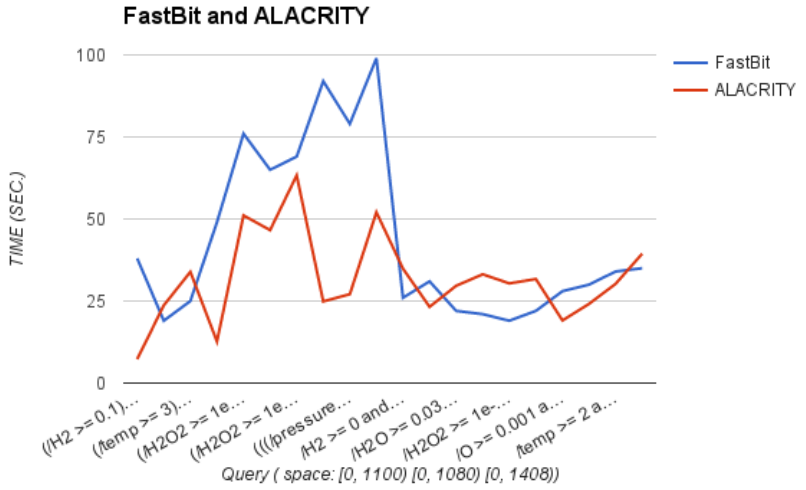

Figure 15: Performance comparison for multivariate query with the *whole* spatial selection.

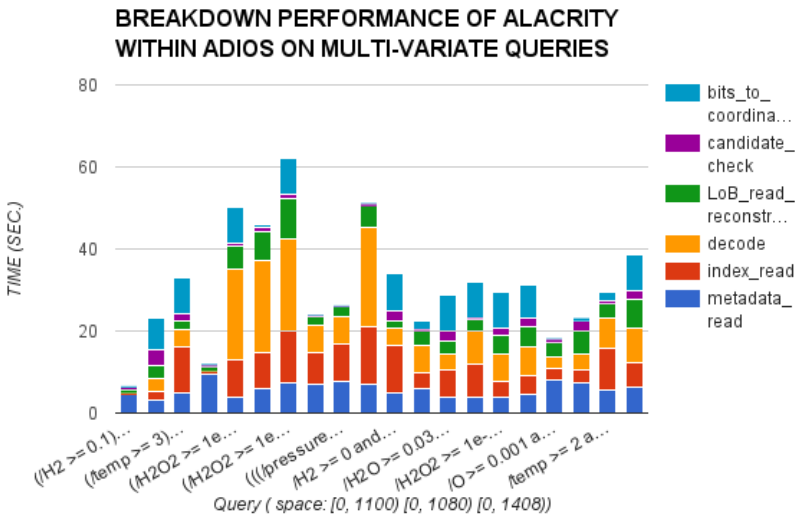

Figure 16: Breakdown performance comparison for multivariate query with the *whole* spatial selection..