Arithmetic Data Cube as a Data Intensive Benchmark

Michael A. Frumkin, Leonid Shabanov
NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center, Moffett Field, CA 94035-1000
frumkin@nas.nasa.gov, leonid.shabanov@crossz.com

January 23 2003

\[ ^1\text{Employee of CrossZ Solutions USA.} \]
Abstract

Data movement across computational grids and across memory hierarchy of individual grid machines is known to be a limiting factor for application involving large data sets. In this paper we introduce the Data Cube Operator on an Arithmetic Data Set which we call Arithmetic Data Cube (ADC). We propose to use the ADC to benchmark grid capabilities to handle large distributed data sets. The ADC stresses all levels of grid memory by producing \(2^d\) views of an Arithmetic Data Set of \(d\)-tuples described by a small number of parameters. We control data intensity of the ADC by controlling the sizes of the views through choice of the tuple parameters.

1 Introduction

The main object of data warehousing and On-Line Analytical Processing (OLAP), decision support database systems, data mining systems and resource brokers is a data set characterized by a number \(d\) of dimension attributes and a measure attribute. The data set consists of tuples \((i_1, \ldots, i_d, c)\). Each dimension attribute \(i_j\) assumes values in some range, say in an interval \([l, m_j - 1]\), and \(c\) is a cost function (a measure) associated with the tuple \((i_1, \ldots, i_d)\). The goal of OLAP is to assist users to discover patterns and anomalies in the data set by providing sort query execution times [13].

A standard tool of OLAP is the Data Cube Operator (DCO) [3], which computes views of the data set. For a chosen subset of \(k\) attributes a view is a set of \(k\)-tuples containing only the chosen attributes with accumulated measures of the duplicates. If technically possible, DCO computes \(2^d\) views on all possible subsets of the dimensions. The calculated DC reduces queries of multi-dimensional data to simple look-ups. There are approaches [1, 5] for mining multi-dimensional association rules and answering iceberg queries by computing an iceberg cube, which contains only aggregates above a certain threshold.

The input data sets and some of the materialized views often do not fit into the main computer memory, thus DCO computation requires a careful reuse of data loaded into the main memory (and all levels of cache). As a rule, computation of the DCO spills data across all levels of memory, making DCO especially interesting as a data intensive benchmark.

A large number of papers is devoted to efficient computation of the DCO [6, 12, 18] and many companies have proprietary algorithms for DCO computations. Some authors propose parallel DCO computation algorithms [8, 10]. To improve the efficiency of querying data cubes a number of publications consider calculation and storage of data cubes as condensed cubes [17] or as other highly compressed structures [14].

We are not trying to evaluate DCO algorithms here, instead we are designing a benchmark for computational grids. For the reference implementation we choose a greedy algorithm [6] that computes each view from the smallest parent (a view having one more attribute). We assume that all attribute values are integers. Although real OLAP data sets and existing OLAP benchmarks [11, 16] use mostly strings as attribute values, this is not a significant limitation, since there are techniques such as hashing for mapping strings to integers. One of the advantages of using integers as attribute values is reduction in the size of the input data sets and materialized views.

Many data sets are available to test OLAP systems, DCO algorithms and data mining algorithms, for example, the ABP-1 and TPC-D benchmark databases [11, 16]. For benchmarking purposes the most appropriate is a synthetic data set which can be generated by a small program, so that the data set will be scalable, the distribution of the benchmark will be manageable, and replication of the data set on the computational grid will incur a small overhead. Also, a synthetic data set, as in many real applications, can be generated in a distributed fashion, which saves the effort of splitting and distributing the data set.

In available synthetic data sets the tuples are randomly generated, however there is no way to control the sizes of the data views. In the next section we introduce the Arithmetic Data Set, which is similar to the randomly generated data sets, but has the advantage of a priori known sizes of the views. The latter simplifies the implementation of the greedy DCO algorithm. For real or random data, one can estimate the view sizes by sampling or analytical methods [6, 15].
2 The Arithmetic Data Set

The purpose of constructing An Arithmetic Data Set is to have a data set whose view sizes can be well controlled. An Arithmetic Data Set $S$ is a subset of a group $Q$ defined by

$$Q = \bigoplus_{i=1}^{d} (\mathbb{Z}/m_i\mathbb{Z})^*,$$

where $(\mathbb{Z}/m_i\mathbb{Z})^*$ is the set of integers modulo $m_i$ coprime with $m_i$. An element of $S$ can be represented by a tuple $x = (x_1, \ldots, x_d)$. The subset $S$ is defined by a seed $s = (s_1, \ldots, s_d) \in Q$, a generator $g = (g_1, \ldots, g_d) \in Q$, $s_i, g_i \neq 0$, $i = 1, \ldots, d$ and the total number of elements $n$:

$$S = \bigcup_{j=0}^{n-1} (s_1 g_1^j, \ldots, s_d g_d^j).$$

We choose $1 \leq g_i < m_i$ to be one of $f_i = |(\mathbb{Z}/m_i\mathbb{Z})^*|$ numbers which are coprime with $m_i$. Let $q_i$ be the order of $g_i$ that is the smallest integer such that $g_i^{q_i} \equiv 1 \bmod(m_i)$. Since $g_i$ can assume at most $f_i$ different values then $g_i^{q_i} \equiv 1 \bmod(m_i)$ and $q_i$ divides $f_i$. These tuples are different elements of $Q$ if $\text{LCM}(q_1, \ldots, q_d) \leq n$, see Corollary 2.

**Data Views.** For any subset containing $k$ of the cube dimensions $I = \{i_1, \ldots, i_k\} \subset \{1, \ldots, d\}$ the $I$-view of $x \in Q$ is defined as a projection of $x$ on the face of the cube defined by $I$:

$$x_I = (x_{i_1}, \ldots, x_{i_k}).$$

The $I$-view of $S$ is comprised of the $I$-view of all elements of $S$:

$$S_I = \{x_I | x \in S\}.$$

**View Sizes.** For a given $I$-view we are interested to find out how many tuples there are in $S_I$. To do this we estimate the *multiplicity* of a tuple $x \in S_I$, defined as the number of tuples of $S$ having the same $I$-view as $x$.

Two tuples $s_I g_I^j$ and $s_I g_I^k$ are the same iff $g_I^j = g_I^k$ or $g_I^{k-j} = 1$ considered as elements of $Q_I$. Hence, the multiplicity $\mu$ of $s_I g_I^j$ can be calculated as follows:

$$\mu = |\{0 \leq k < n \mid k - j \equiv 0 \bmod(q_i), i \in I\}|.$$

Since the smallest nonzero solution of the system of congruences $k - j \equiv 0 \bmod(q_i)$, $i \in I$, is $\lambda_I = \text{LCM}_{i \in I}(q_i)$ we find that $\left\lfloor \frac{n}{\lambda_I} \right\rfloor \leq \mu \leq \left\lfloor \frac{n}{\lambda_I} \right\rfloor + 1$, which proves the following assertion.

**PROPOSITION 1.** Let $\lambda_I = \text{LCM}_{i \in I}(q_i)$. The multiplicity $\mu$ of any tuple of an $I$-view of $S$ can be estimated as

$$\left\lfloor \frac{n}{\lambda_I} \right\rfloor \leq \mu \leq \left\lfloor \frac{n}{\lambda_I} \right\rfloor + 1.$$

If $\lambda_I > n$, then the second inequality of the proposition implies that multiplicity of each element of $S_I$ is 1, hence $|S_I| = n$. Obviously, $|S_I| \leq \lambda_I$. Hence, we have the following formula for $|S_I|$:

**COROLLARY 2.** For the size of an $I$-view of $S$ we have the following relation:

$$|S_I| = \begin{cases} n, & \text{if } n \leq \lambda_I; \\ \lambda_I, & \text{otherwise.} \end{cases}$$

\(^1\text{LCM stands for the Least Common Multiple.}\)
3 Choice of the Parameters

To illustrate a possible choice of the parameters for the grid benchmarks we choose \( m_i \) to be prime numbers and \( g_i \) to be generators of \((\mathbb{Z}/m_i\mathbb{Z})^*\), hence having period \( q_i = f_i = m_i - 1 \). Also, we choose \( m_i \) such that \( m_i - 1 \) has many small prime factors so that \( \lambda_f \) has a good chance of being small. This approach gives us good control over the sizes of the data set and its views. Our actual choice of the \( m_i \) is shown in the Table 1.

We choose 4 groups of the smallest prime numbers \{3, 5, 7\}, \{11, 13, 17, 19\}, \{23, 29, 31, 37\}, and \{41, 43, 47, 53, 59\}. For each group we choose 5 smallest primes \( m_i \) such that prime factors of \( m_i - 1 \) are 2 and numbers from this group. This set of parameters gives us a data set of \( 2^5 \cdot 3^2 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19^2 \cdot 23 \cdot 29 \cdot 31^2 \cdot 37 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 \) different tuples and, for example we can choose \( n = 2 \cdot 11 \cdot 23 \cdot 41 \cdot 3 \cdot 13 \cdot 29 \cdot 43 \cdot 5 \cdot 17 = 85759918530 \). At the same time the 5-dimensional views relative to each of the groups is small relative to the number of the total elements in the data set.

Table 1. Dimensions of the Arithmetic Data Cube and their factorizations. Here "Least Generator" \( \gamma_i \) is the smallest generator of \((\mathbb{Z}/m_i\mathbb{Z})^*\), the "Generator" is the chosen generator of \((\mathbb{Z}/m_i\mathbb{Z})^*\) and the "Exp" is \( e_i \) such that \( g_i = \gamma_i^{e_i} \).

<table>
<thead>
<tr>
<th>Prime</th>
<th>Factorization of ( m - 1 )</th>
<th>Least Generator</th>
<th>Exp</th>
<th>Generator</th>
<th>Seed ((m + 1)/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 421</td>
<td>( 2^2 \cdot 3 \cdot 5 \cdot 7 )</td>
<td>2</td>
<td>11</td>
<td>364</td>
<td>211</td>
</tr>
<tr>
<td>2. 601</td>
<td>( 2^3 \cdot 3 \cdot 5 \cdot 7 )</td>
<td>7</td>
<td>13</td>
<td>412</td>
<td>301</td>
</tr>
<tr>
<td>3. 631</td>
<td>( 2 \cdot 3^2 \cdot 5 \cdot 7 )</td>
<td>3</td>
<td>17</td>
<td>334</td>
<td>316</td>
</tr>
<tr>
<td>4. 701</td>
<td>( 2^2 \cdot 5 \cdot 7 )</td>
<td>2</td>
<td>19</td>
<td>641</td>
<td>351</td>
</tr>
<tr>
<td>5. 883</td>
<td>( 2 \cdot 3^2 \cdot 7 )</td>
<td>2</td>
<td>23</td>
<td>108</td>
<td>442</td>
</tr>
<tr>
<td>LCM</td>
<td>( 2^3 \cdot 3^2 \cdot 5^2 \cdot 7^2 = 88200 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. 419</td>
<td>( 2 \cdot 11 \cdot 19 )</td>
<td>2</td>
<td>23</td>
<td>228</td>
<td>210</td>
</tr>
<tr>
<td>7. 443</td>
<td>( 2 \cdot 13 \cdot 17 )</td>
<td>2</td>
<td>29</td>
<td>98</td>
<td>222</td>
</tr>
<tr>
<td>8. 647</td>
<td>( 2 \cdot 17 \cdot 19 )</td>
<td>5</td>
<td>31</td>
<td>94</td>
<td>324</td>
</tr>
<tr>
<td>9. 21737</td>
<td>( 2^3 \cdot 11 \cdot 13 \cdot 19 )</td>
<td>31</td>
<td>37</td>
<td>8280</td>
<td>10869</td>
</tr>
<tr>
<td>10. 31769</td>
<td>( 2^3 \cdot 11 \cdot 19^2 )</td>
<td>7</td>
<td>41</td>
<td>26667</td>
<td>15885</td>
</tr>
<tr>
<td>LCM</td>
<td>( 2^3 \cdot 11 \cdot 13 \cdot 17 \cdot 19^2 = 7020728 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. 1427</td>
<td>( 2 \cdot 23 \cdot 31^2 )</td>
<td>2</td>
<td>41</td>
<td>595</td>
<td>714</td>
</tr>
<tr>
<td>12. 18353</td>
<td>( 2^3 \cdot 31 \cdot 37 )</td>
<td>3</td>
<td>43</td>
<td>8397</td>
<td>9177</td>
</tr>
<tr>
<td>13. 22817</td>
<td>( 2^5 \cdot 23 \cdot 31 )</td>
<td>3</td>
<td>47</td>
<td>15046</td>
<td>11409</td>
</tr>
<tr>
<td>14. 34337</td>
<td>( 2^6 \cdot 29 \cdot 37 )</td>
<td>3</td>
<td>53</td>
<td>15699</td>
<td>17169</td>
</tr>
<tr>
<td>15. 98717</td>
<td>( 2^3 \cdot 23 \cdot 29 \cdot 37 )</td>
<td>2</td>
<td>59</td>
<td>62206</td>
<td>49359</td>
</tr>
<tr>
<td>LCM</td>
<td>( 2^6 \cdot 23 \cdot 29 \cdot 31^2 \cdot 37 = 758228608 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16. 3527</td>
<td>( 2 \cdot 41 \cdot 43 )</td>
<td>5</td>
<td>3</td>
<td>125</td>
<td>1764</td>
</tr>
<tr>
<td>17. 8693</td>
<td>( 2^3 \cdot 41 \cdot 53 )</td>
<td>3</td>
<td>5</td>
<td>443</td>
<td>4347</td>
</tr>
<tr>
<td>18. 9677</td>
<td>( 2^2 \cdot 41 \cdot 59 )</td>
<td>2</td>
<td>7</td>
<td>128</td>
<td>4839</td>
</tr>
<tr>
<td>19. 11093</td>
<td>( 2^2 \cdot 47 \cdot 59 )</td>
<td>2</td>
<td>11</td>
<td>2048</td>
<td>5547</td>
</tr>
<tr>
<td>20. 18233</td>
<td>( 2^3 \cdot 43 \cdot 53 )</td>
<td>3</td>
<td>13</td>
<td>8052</td>
<td>9117</td>
</tr>
<tr>
<td>LCM</td>
<td>( 2^3 \cdot 41 \cdot 43 \cdot 47 \cdot 53 \cdot 59 = 2072850776 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 Air Traffic Control Example

We illustrate the Data Cube Operator for an example of Air Traffic Control (ATC) data [9]. Each of about 20 national ATC Centers obtain flight data from airports and radars in real time. Typical records are shown in Table 2 and a typical query is as follows:

Since we use odd primes, \( m_i - 1 \) always has 2 as a factor
Find AC type
where Busy = 1
and ETA is Between 1105 and 1110
and destination is CLE

The queries should be executed in real time and can be posted at any of the centers, implying that the flight data must be communicated between the centers. One possible way to insure a short query response time is to replicate the Data Cube across the centers. This constitutes an example of a distributed dynamic DCO which requires real time query response and DCO update. The ATC example can be extended to an example of Satellite and Spacecraft Control system.

Table 2. Air Traffic Control Data. Typical Query: Find AC type where Busy = 1 and ETA is between 1105 and 1110 and destination is CLE.

<table>
<thead>
<tr>
<th>Flight ID</th>
<th>AC type</th>
<th>ETA</th>
<th>Destination</th>
<th>Controller</th>
<th>Busy</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAL 147</td>
<td>747</td>
<td>1100</td>
<td>CLE</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>NW 1186</td>
<td>767</td>
<td>1132</td>
<td>ORD</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>KLM 761</td>
<td>747</td>
<td>1105</td>
<td>CLE</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>AA 2345</td>
<td>A320</td>
<td>1135</td>
<td>ORD</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>UAL 258</td>
<td>737</td>
<td>1112</td>
<td>CLE</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>AA 2744</td>
<td>737</td>
<td>1105</td>
<td>CAK</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>SW 377</td>
<td>767</td>
<td>1108</td>
<td>CLE</td>
<td>87</td>
<td>1</td>
</tr>
</tbody>
</table>

5 Related Work

The benchmarking of data mining systems is well established area of High Performance Computing [11, 16]. These benchmarks are designed to compare performance of query systems running on a server. On the other hand, a number of benchmarks have been designed for testing computational grids [2, 11, 16]. The grid benchmarking effort is currently supported by the Grid Benchmarking Research Group at the Global Grid Forum. These benchmarks are mostly computationally intensive and are derived from NAS Parallel Benchmarks. We propose the Arithmetic Data Cube (ADC) as a data intensive grid benchmark which extends typical data mining operations into a grid environment.

6 Conclusions

We show that ADC represents an important set of computations in the OLAP and data mining. We give an example of a dynamic real time system performing the set of operations specified in ADC. The ADC is data intensive since

- it mostly involves logical operations
- the size of the output data set can significantly exceed the size of the original data set
- existing algorithms perform few operations per tuple per memory access (and are similar to the merge in this respect)

The advantages of ADC as a grid benchmark are that

- it is described by a small number of parameters and has a priori known sizes of the views
- the views can be generated independently
- the overhead of combining the views is predictable
the data set can be partitioned into a number of independently generated subsets

- the elements of the data set are pseudo random

These two properties make ADC a strong candidate for a data intensive grid benchmark to be considered by the Global Grid Forum Grid Benchmarking Research Group (GB-RG) [4].
Bibliography


