Fast In-Memory SQL Analytics on Typed Graphs*

Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, Matthias Springer
Department of Computer Science and Engineering, University of California, San Diego
{chunbinlin, bmandel, yannis, mspringer}@cs.ucsd.edu

ABSTRACT

We study a class of graph analytics SQL queries, which we call *relationship queries*. These queries involving aggregation, join, semijoin, intersection and selection are a wide superset of fixed-length graph reachability queries and of tree pattern queries. We present real-world OLAP scenarios, where efficient relationship queries are needed. However, row stores, column stores and graph databases are unacceptably slow in such OLAP scenarios.

We propose a *GQ-Fast* database, which is an indexed database that roughly corresponds to efficient encoding of annotated adjacency lists that combines salient features of column-based organization, indexing and compression. GQ-Fast uses a bottom-up fully pipelined query execution model, which enables (a) aggressive compression (e.g., compressed bitmaps and Huffman) and (b) avoids intermediate results that consist of row IDs (which are typical in column databases). GQ-Fast compiles query plans into executable C++ source code. Besides achieving runtime efficiency, GQ-Fast also reduces main memory requirements because, unlike column databases, GQ-Fast selectively allows dense forms of compression including heavy-weight compressions, which do not support random access.

We used GQ-Fast to accelerate queries for two OLAP dashboards in the biomedical field. GQ-Fast outperforms PostgreSQL by 2–4 orders of magnitude and MonetDB, Vertica and Neo4j by 1–3 orders of magnitude when all of them are running on RAM. Our experiments dissect GQ-Fast's advantage between (i) the use of compiled code, (ii) the bottom-up pipelining execution strategy, and (iii) the use of dense structures. Other analysis and experiments show the space savings of GQ-Fast due to the appropriate use of compression methods. We also show that the runtime penalty incurred by the dense compression methods decreases as the number of CPU cores increases.

1. INTRODUCTION

*This work was supported by NSF 1447943.
We thank the anonymous reviewers for their constructive comments. Furthermore, we are grateful to Benjamin Good from the Scripps Research Institute for valuable discussions and feedback.

Proceedings of the VLDB Endowment, Vol. 10, No. 3

The focus of past OLAP systems was on SQL queries on data cubes, whose data is modeled as star/snowflake SQL schemas [9, 36, 14]. However, in recent years, an avalanche of graph data emerged, such as disease-drug networks (chem/bio-informatics) [20, 21] and social networks (Web) [38, 39]. A new generation of benchmarks, such as the Microsoft Academic Graph (MAG) Benchmark [34] and the Berkeley Big Data Benchmark [33] make clear the distinction of these data from data cubes (such as the old TPC-H benchmark). The particular data sets and benchmarks, as well as many others, are essentially *typed graphs*, i.e., graphs where vertices and edges are associated with types known in advance. There is an increasing demand to perform analytic SQL queries over such graphs; e.g., discovering related diseases in a disease-drug network graph. Traditional, SQL OLAP technologies do not handle such demands well because they are not sufficiently optimized for finding paths among entities [11, 10].

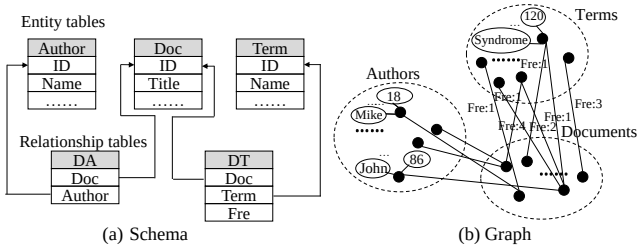

Figure 1: PubMed Schema and Corresponding Graph. Each entity table corresponds to a type of vertices, while each relationship table corresponds to edges linking corresponding types of vertices.

Schema. Towards SQL-based OLAP on graphs, we first define the representation of *typed graphs* (also known as graphs with schema, e.g., [18]) in an SQL database. The nodes and edges of a typed graph are represented as tuples of relational tables. We classify the tables into two categories: *Entity tables* and *Relationship tables*, following the database E/R model [15]. We focus on *binary* relationships. Each entity table has a primary key column, called the *ID* column, while each relationship table has two foreign key columns pointing to ID columns of entity tables[1]. Hence, the tuples comprise a typed graph [37, 41]: Each entity table corresponds to a type of vertices, while each relationship table corresponds to a type of edges. Columns in entity tables and relationship tables correspond to attributes of vertices and edges, respectively. For example, consider the premier public biomedical database PubMed[2]. Figure 1(a)

[1]In order to capture many-to-one relationships efficiently, we also allow entity tables to have foreign keys [15]. We neglect this possibility, as it does not essentially change any consideration.
[2]http://www.ncbi.nlm.nih.gov/pubmed

shows its schema, and Figure 1(b) presents a corresponding typed graph. The tuples of the relationship table DT stand for edges from a document tuple/entity/node to a term tuple/entity/node.

Relationship Queries. We identify a class of queries, called *relationship queries*, which cover many analytics needs on graph data and, in addition, they are amenable to orders-of-magnitude speed optimization. Informally, a relationship query contains three steps: (i) *Context Computation*: The context is a collection of entities whose properties satisfy the user given conditions; (ii) *Path Navigation*: Navigation from source entities to target entities is via joins over relationship tables; and (iii) *Path Aggregation*: The importance of the target entities is computed by applying aggregation functions over attributes collected along the navigation paths, which are accessed in the first step. The first and third steps are optional. Relationship queries are common in graph analytics. For example, all the queries evaluated in [26] are relationship queries. We illustrate a relationship query on the PubMed schema, which will serve as one of the running examples.

Query SD (Similar Documents). Assume a user wants to find documents d_j that are similar to a given document d_0 with ID d_0^{ID} in the PubMed graph. Similarity between documents d_0 and d_j is measured by the number of terms associated to both of them, i.e., the number of paths with type Doc → Term → Doc that start at d_0 and end at d_j.

```
SELECT dt2.Doc, COUNT(*) AS similarity
FROM DT dt1 JOIN DT dt2 ON dt1.Term = dt2.Term
WHERE dt1.Doc = d0^ID
GROUP BY dt2.Doc
```

The Query SD is a simple relationship query: It navigates via typed paths Doc → Term → Doc and then aggregates the number of paths reaching each target. More complex (and performance-challenging) relationship queries are presented in Section 2.

It is challenging to answer even this simple query efficiently, due to the large size of the graph: thirty million vertices and one billion edges with several attributes. Given the analytical nature of relationship queries, column-oriented database systems are much more efficient than row-stores and graph database systems, as our experiments verified (see Section 6) [35, 3, 2, 19]. Nevertheless, the obtained performance is often insufficient for online queries and interactive applications. Query SD takes 61.6 and 19.17 seconds on the column databases MonetDB [19] and Vertica [23], 741.2 seconds on the row database PostgreSQL, and 49.3 seconds on the graph database Neo4j, even though we fully cached the data in main memory in all the cases. Performance gets far worse when the join paths are longer, the aggregations involve many attributes of the paths or the source entities themselves are specified by their properties and connections, rather than their IDs.

To improve the performance, we propose an index-only fully pipelined database called GQ-Fast. GQ-Fast answers Query SD in 1.068 seconds. As the queries become more complex, its performance ratio to the other systems widens. Moreover, GQ-Fast generally requires less memory.

GQ-Fast achieves such superior performance by employing a code generator to produce efficient fully pipelined source code running upon a new compressed fragment-based index, as outlined in the following paragraphs.

Database Structure. A GQ-Fast database physically stores only indices – it does not store the logical tables. Generally, the administrator may load a relation $R(C_1, C_2, ..., C_n)$ and specify that for each ID or foreign key attribute C a respective index should be built, using C as the *indexed column*. In response, GQ-Fast will make an index $\mathcal{I}_{R.C}$ for each such attribute. Figure 2 shows the two indices $\mathcal{I}_{\text{DT.Doc}}$ and $\mathcal{I}_{\text{DT.Term}}$ that correspond to the two

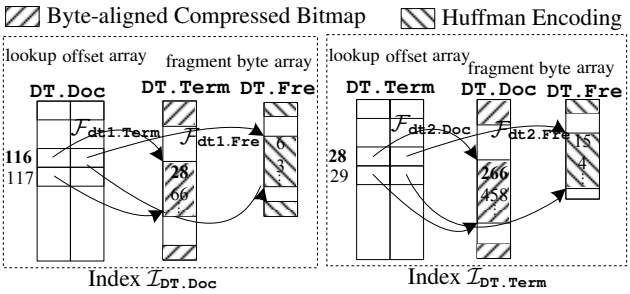

Figure 2: Example of Fragments and Query Processing for Query SD. Fragments $\pi_{Term}\sigma_{Doc=116}DT$ and $\pi_{Fre}\sigma_{Doc=116}DT$ are encoded with *bit-aligned compressed array* and *Huffman encoding*.

foreign keys of the table DT. During runtime, the GQ-Fast query processor will use the index to find (projections of) tuples of R that have a given $C = c$ value. For example, the index $\mathcal{I}_{\text{DT.Doc}}$ can be used to find the terms associated to document 116 (i.e., $\pi_{\text{Term}}\sigma_{\text{Doc}=116}\text{DT}$) or to find the term/frequency pairs associated to document 116 (i.e., $\pi_{\text{Term, Fre}}\sigma_{\text{Doc}=116}\text{DT}$).

Internally, a GQ-Fast index has two components: a lookup table and a set of fragments. Let us say, w.l.o.g., that C_1 is the indexed column. Then for each column $C_j \in \{C_2, \ldots, C_n\}$ and for each value $t \in C_1$ there is a fragment $\pi_{C_j}\sigma_{C_1=t}(R)$, which retains the original order of the values. In Figure 2, the fragment $\pi_{\text{Term}}\sigma_{\text{Doc}=116}\text{DT}$ (with contents 28, 66, etc.) and the fragment $\pi_{\text{Fre}}\sigma_{\text{Doc}=116}\text{DT}$ (with contents 6, 3, etc.) provide the terms and frequencies associated to document 116, respectively.

To reduce space costs, GQ-Fast compresses individual fragments. *The key observation behind compressing fragments is that when a relationship query accesses a fragment, all of its data will be used. There is no need for random access within the fragment.* Based on this, GQ-Fast allows very compressed encodings of each individual fragment, such as Huffman encoding. Note that the typical fragment is relatively small (compared to the column) and can typically fit in the L1 cache or, at least, in the L2 cache. Hence, its decoding is not penalized with multiple random access to the RAM.

Given a value for the indexed column, the lookup table must be able to provide a pointer to the respective fragment, along with the size of each fragment. A lookup table can be built in many known ways; e.g., as a hash table. GQ-Fast saves space and response time by building *lookup tables as offset arrays that utilize the dense ID assumption*, according to which the IDs of an entity table are consecutive integers, starting from 0. Under this assumption, all fragments of the same type are listed consecutively in an array: A GQ-Fast lookup table for an index on R.C_1 is a two-dimensional array $\mathcal{I}_{\text{R.C}_1}$ of size $v \times (n-1)$, where v is the number of unique values in R.C_1 and n is the number of columns in R. The starting address of the fragment $\pi_{C_j}\sigma_{C_1=t}(R)$ is stored in $\mathcal{I}_{\text{R.C}_1}[t][j-1]$ and its size can be calculated using the starting address of the next fragment.

Query Processing. GQ-Fast query plans run exclusively on indices. They employ a *bottom-up pipelined execution model*, illustrated next, to avoid large intermediate results. In addition, GQ-Fast employs a C++ code generator for query plans.

As an example, consider the generated code for Query SD, which uses table DT with columns Document (0^{th} column) and Term (1^{st} column). In Lines 2–4, GQ-Fast uses index $\mathcal{I}_{\text{DT.Doc}}$ to find the Terms' fragment $\pi_{\text{Term}}\sigma_{\text{Doc}=116}$ $(\text{DT} \mapsto \text{dt1})$ starting at position $\mathcal{I}_{\text{DT.Doc}}$ $[116][1]$ with size $l_{\text{dt1.Term}}$. GQ-Fast decodes the fragment into the (preallocated) array $\mathcal{A}_{\text{dt1.Term}}$ and returns the number of elements $n_{\text{dt1.Term}}$. Afterwards (Lines 5–9), for each

Generated Code: Generated code for Query SD

```
1  R ← ∅
2  F_dt1.Term ← I_DT.Doc[116][1]
3  l_dt1.Term ← I_DT.Doc[116 + 1][1] − F_dt1.Term
4  A_dt1.Term, n_dt1.Term ← decodeBB(F_dt1.Term, l_dt1.Term)
5  for i ← 0 to n_dt1.Term − 1 do
6      v_dt1.Term ← A_dt1.Term[i]
7      F_dt2.Doc ← I_DT.Term[v_dt1.Term][0]
8      l_dt2.Doc ← I_DT.Term[v_dt1.Term + 1][0] − F_dt2.Doc
9      A_dt2.Doc, n_dt2.Doc ← decodeBB(F_dt2.Doc, l_dt2.Doc)
10     for j ← 0 to n_dt2.Doc − 1 do
11         v_dt2.Doc ← A_dt2.Doc[j]
12         R[v_dt2.Doc] ← R[v_dt2.Doc] + 1
13 return R
```

term ID $v_{\mathrm{dt1.Term}} \in \mathcal{A}_{\mathrm{dt1.Term}}$, GQ-Fast uses index $\mathcal{I}_{\mathrm{DT.Term}}$ to find the Documents fragment $\pi_{\mathrm{dt2.Doc}}\sigma_{\mathrm{dt2.Term}=v_{\mathrm{dt1.Term}}}$ ($\mathrm{DT}\mapsto \mathrm{dt2}$) starting at position offset $\mathcal{I}_{\mathrm{DT.Term}}[v_{\mathrm{dt1.Term}}][0]$ (see Figure 2). GQ-Fast decodes the identified fragments into $\mathcal{A}_{\mathrm{dt2.Doc}}$. Finally, GQ-Fast scans all Documents fragments to update the array $\mathcal{R}$ (Lines 11–12), which holds the counts per document.

Notice that (1) GQ-Fast can afford to have $\mathcal{R}$ be an array (as opposed to a hash table) because of the dense ID assumption; (2) The execution is pipelined in a sense that it iterates over the fragments and their elements. The memory footprint is small as it is dictated by the max. size of fragments and not of the overall column size.

Contributions. This paper makes the following contributions.

- Formally identifies *relationship queries*, a subset of SQL that is both important and amenable to orders-of-magnitude optimization. We illustrate the subset's importance with (a) examples from two real-world use cases (Section 2) and (b) by providing their syntax, showing that it captures a very large part of SQL.
- Describes a new *index-only and fragment-based data organization* (Section 4) and coordinated query plans (Section 5). The bottom-up query plan enables (a) aggressive compression (e.g., compressed bitmaps and Huffman) and (b) avoids intermediate results that consist of row IDs (which are typical in column databases). Further space savings and speed improvements stem from using a *dense* (consecutive) entity IDs assumption.
- Describes a code generator that produces C++ code for each query plan. The produced code benefits from CPU-level optimizations (Section 5).
- Performs experiments on three real-life datasets (PubMed-M, PubMed-MS [20] and SemMedDB [21]), showing GQ-Fast is $10 - 10^4$ times more efficient than MonetDB, Neo4j, Vertica and PostgreSQL when all are running on main memory (Section 6). Since the performance advantage is due to many factors, a comprehensive series of experiments isolates the marginal effect of each individual factor/optimization.

GQ-Fast has been deployed in two real world use cases around PubMed and SemMedDB data, has been released online, and is behind a publicly accessible interactive demo system[3].

2. DATA SCHEMA AND QUERIES

2.1 Data Schema

We classify relational tables in GQ-Fast in two categories according to the entities and the relationships of the E/R model [15]: *entity tables* (e.g., Author in Figure 1(a)) and *relationship tables* (e.g., DT, DA). Each entity table $\mathbb{E}$ has an ID (primary key) attribute and several attributes $M_1, ..., M_n$. Each tuple $t \in \mathbb{E}$ corresponds to a real-life entity. A relationship table $\mathbb{R}$ has two *foreign key attributes* F_1 and F_2 referencing the IDs of respective entity tables[4], i.e., $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$, where $\rightsquigarrow$ is *reference*. The combination $(f_1, f_2) \in F_1 \times F_2$ is unique. A relationship table may also have *measure attributes* $M_1, ..., M_m$ (e.g., DT.Fre).

E/R → Graph. Mapping E/R schemas to graph models is a well-studied topic [37, 7]. We use the following two steps to map our schema to a typed graph [18]: (i) Each entity table $\mathbb{E}(M_1, ..., M_n)$ refers to a type of vertices $\mathbb{V}$, and each entity $t \in \mathbb{E}$ refers to one vertex $v \in \mathbb{V}$. The attributes $M_1, ..., M_n$ in the entity table are mapped to properties of vertices; and (ii) each relationship table $\mathbb{R}(F_1, F_2, M_1, ..., M_m)$ refers to edges $\mathbb{E}$ crossing two types of vertices $\mathbb{V}_1 \times \mathbb{V}_2$, where $\mathbb{V}_1$ and $\mathbb{V}_2$ are translated from entities $\mathbb{E}_1$ and $\mathbb{E}_2$ and $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$.

Graph → Relational Schema with Entities and Relationships. Mapping a graph to a relational schema has been studied for several years [12, 40]. We first show how to convert a typed graph to our E/R schema, then describe general graphs. Mapping a typed graph to our schema has the following two steps: First, store vertices of the same type into one entity table. Each attribute of the vertices becomes one column in the table. Second, store edges that have the same type into the same relationship table. Edges of the same type have source (resp. target) nodes that have the same type.

For general graphs, the basic way is to store all the vertices in one big *Node(ID, Type)* table, while all the edges are in one *Edge(Source, Destination, Type)* table. If more detailed knowledge about the types of vertices can be inferred from the graph, then the mapping approach of typed graphs can be more fine-grained.

2.2 Relationship Query

Informally, a relationship query proceeds in three steps: (i) *Context Selection:* Entities satisfying query conditions (i.e., certain user-provided properties) are marked as a context; (ii) *Path Navigation:* To reach target entities from the context, queries "navigate" between entities via join operations; and (iii) *Relevance Computation:* The relevance between each target entity and the context is computed by applying aggregation functions over measure attributes collected in the second step.

In its algebraic form, a relationship query involves σ (selection), π (projection), $\bowtie$ (join), $\ltimes$ (semi-join) operators and an optional γ (aggregation) at the end, and must satisfy the follow restrictions: (i) join and semijoin conditions are equalities between (primary or foreign) key attributes and (ii) aggregations group-by on a primary key or foreign key. The set of relationship queries includes graph reachability (path finding) queries, where the edges are defined by foreign keys. More generally, it includes tree pattern queries, followed by aggregation. The first restriction does not narrow down the scope of relationship query applications as it only requires that navigation on a graph should be performed via connected edges, which is a natural requirement for graph navigation. The second restriction allows GQ-Fast to use an array to maintain the aggregation results instead of using a map, which contributes to 30% performance improvement (see Table 9 in Section 6.2.3).

Example Queries. We now illustrate a number of relationship queries using the datasets of some GQ-Fast applications: PubMed

[3] http://137.110.160.52:8080/GQFast-System

[4] In this paper, we focus on relationship tables with two foreign keys.

and SemMedDB. These queries were used in our experiments and are implemented in our interactive demo system.

Even though the definition of relationship queries includes a larger set of queries, we focus on these queries, because they illustrate accurately the use cases for which GQ-Fast was designed and achieves the best speedup compared to other database systems: queries with long join paths involving many-to-many relationships. In the following examples, we use $E_1 \rightarrow E_2$ to visualize a join from table E_1 to table E_2 and $\circlearrowright_E$ to visualize an intersection on table E.

2.2.1 Example Queries in PubMed

FSD (Frequency-Time-aware Document Similarity). Query FSD computes time-aware and frequency-aware cosine similarity. The cosine similarity is computed as follows: Each document d is associated with a vector $t^d = [t_1^d, \ldots, t_n^d]$, where n is the number of terms across all documents. The cosine similarity between two documents x and y is defined as $\sum_{i=1,\ldots,n} t_i^x t_i^y$[5] In contrast to Query SD in the Introduction, Query FSD raises the similarity degree of documents that are chronologically close. The navigation path of Query FSD can be visualized as d1 → dt1 → dt2 → d2.

```
SELECT dt2.Doc, SUM(dt1.Fre * dt2.Fre)/(abs(d1.Year-d2.Year)+1)
FROM (((Doc d1 JOIN DT dt1 ON d1.ID = dt1.Doc)
      JOIN DT dt2 ON dt1.Term = dt2.Term)
     JOIN Doc d2 ON d2.ID = dt2.Doc)
WHERE d1.ID = d_0^ID
GROUP BY dt2.Doc
```

AD (Authors' Discovery). Query AD finds the authors who published papers that pertain to the terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ (e.g., authors that published papers related to the terms "neoplasms" and "statins") and counts the number of papers per author. The navigation path of Query AD can be visualized as $\circlearrowright_{dt}$→da.

```
SELECT da.Author, COUNT(*)
FROM DA da
WHERE da.Doc IN
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = t_1^ID)
  INTERSECT
  ...
  INTERSECT
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = t_n^ID)
GROUP BY da.Author
```

FAD (Co-Occurring Terms Discovery). Query FAD is similar to Query AD. It finds other terms that co-occur in documents about terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ along with the number of occurrences (e.g., terms that co-occur in documents about "neoplasms" and "statins" and how often). The navigation path of Query FAD can be visualized as $\circlearrowright_{dt}$→dt1.

```
SELECT dt1.Term, Sum(dt0.Fre)
FROM DT dt1
WHERE dt.Doc IN
  (SELECT dt.Doc FROM DT dt1 WHERE dt1.Term = t_1^ID)
  INTERSECT
  ...
  INTERSECT
  (SELECT dt.Doc FROM DT dtn WHERE dtn.Term = t_n^ID)
GROUP BY dt1.Term
```

AS (Author Similarity). Query AS finds the authors whose publications relate to the Mesh terms in the publications of a given author, identified by the id a^{ID}. Furthermore, each discovered author is given a weight/similarity score by first computing the similarity of the publications using the cosine of the term frequencies and then weighing recent publications heavier. The navigation path of Query AS can be visualized as da1→dt1→dt2→d→da2.

```
SELECT da2.Author,SUM(dt1.Fre × dt2.Fre)/(2017-d.Year)
FROM (((DA da1 JOIN DT dt1 ON da1.Doc=dt1.Doc)
       JOIN DT dt2 ON dt1.Term = dt2.Term)
      JOIN Doc d ON dt2.Doc=d.ID)
JOIN DA da2 ON dt2.Doc=da2.Doc
WHERE da1.Author = a^ID
```

[5] In practice, the queries also normalize for the sizes of t^x and t^y and, in later examples, the sizes of measures. The examples exclude the normalization since they do not present any important additional aspect to the exhibited query pattern.

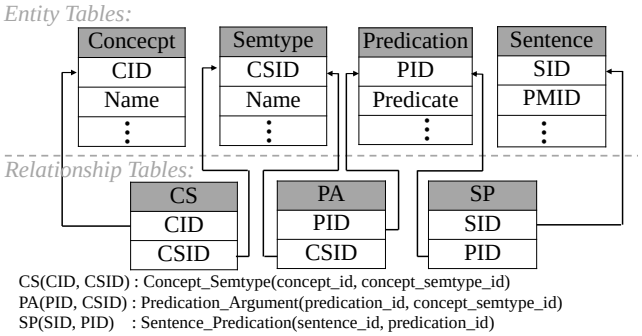

Figure 3: SemMedDB Database Schema. CS, PA and SP are relationship tables, whereas the others are entity tables.

2.2.2 Example Queries in SemMedDB

The Scripps Research Institute implemented *Knowledge.Bio*[6], a system for exploring, learning, and hypothesizing relationships among concepts of the SemMedDB database[7], which is a repository of semantic predications (subject-predicate-object triples). Figure 3 shows the schema of SemMedDB.

CS (Concept Similarity). As a use case of Knowledge.Bio, Query CS finds the concepts that are most relevant to a given concept, e.g., "Atropine", where c^{ID} is the concept ID of "Atropine". The navigation path of Query CS can be visualized as c1 → p1 → s1 → s2 → p2 → c2.

```
SELECT c2.CID, COUNT(*)
FROM CS c2, PA p2, SP s2
WHERE s2.PID = p2.PID
  AND p2.CSID = c2.CSID AND s2.SID IN
    (SELECT s1.SID
     FROM CS c1, PA p1, Sp s1
     WHERE s1.PID = p1.PID AND p1.CSID = c1.CSID
       AND c1.CID = c^ID)
GROUP BY CID
```

The running time of this query on an Amazon Relational Database Service (Amazon RDS) with MySQL was 25 minutes. GQ-Fast reduced the running time for that query to less than 1 second.

3. ARCHITECTURE

Applications use GQ-Fast as an OLAP-oriented database that accompanies their original transaction-oriented databases. Figure 4 gives an overview of GQ-Fast's architecture. It has two parts: *GQ-Fast Database Generation* and *GQ-Fast Query Processing*.

GQ-Fast Database Generation. The *GQ-Fast Loader* receives loading commands, retrieves data from one or multiple relational databases, and creates GQ-Fast indices along with relevant metadata, containing information about fragments and their encodings. This phase is done offline. The schema of the GQ-Fast database has to follow certain conventions (see Section 2). GQ-Fast data is stored in main memory data structures (see Section 4).

[6] http://knowledge.bio/

[7] http://skr3.nlm.nih.gov/SemMedDB/dbinfo.html

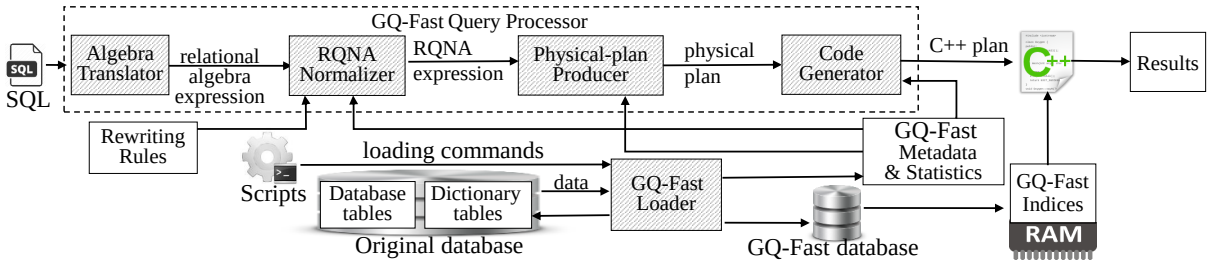

Figure 4: Architecture of Relationship Query Processing in GQ-Fast. The *GQ-Fast Loader* produces the GQ-Fast database and metadata, and the *GQ-Fast Query Processor* generates code answering a given relationship query.

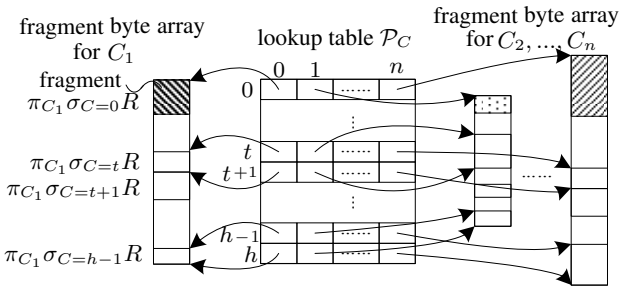

Figure 5: Index $\mathcal{I}_{R.C}$. The lookup table $\mathcal{P}_c$ stores offsets of fragments. Fragments of different columns have different encodings.

When loading data into GQ-Fast, users should specify (i) the columns to be indexed, upon which GQ-Fast builds lookup tables. Then, GQ-Fast organizes the values in other columns as fragments; and (ii) an encoding method for each column excluding indexed columns. Section 4 provides detailed guidelines for choosing proper encoding methods for different columns.

GQ-Fast Query Processing. The *GQ-Fast Query Processor* receives an SQL query and outputs its result. It consists of several subcomponents. The *Algebra Translator* translates an SQL query into a relational algebra expression, which is then transformed into a *Relationship Query Normalized Algebra (RQNA) expression* (see Section 5.1) by the *RQNA Normalizer* using rewriting rules. The *RQNA Normalizer* also verifies whether an SQL query is a relationship query by checking the restrictions according to metadata. Afterwards, the *Physical-plan Producer* transforms the RQNA expression into a physical-level plan. The *Code Generator* consumes the physical plan and metadata and produces C++ code, which is then compiled and ran on the GQ-Fast index to get final results. GQ-Fast can also prepare a query statement, and then execute it multiple times (as JDBC does), changing the parameters each time.

4. GQ-FAST INDEX STRUCTURE

This section first presents the GQ-Fast index structure and analyzes different encoding methods, then describes how to build indices for both entity and relationship tables. Finally, a discussion of how to support incremental updates is provided.

Index Structure. Given a relation $R(C, C_1, C_2, ..., C_n)$, assuming the indexed column is C, GQ-Fast builds one index $\mathcal{I}_{R.C}$ (shown in Figure 5) for R. A GQ-Fast index has one lookup table for the indexed column C, and organizes values in columns $C_1, ..., C_n$ in fragments. We assume that $|R| = h$, consequently the column C contains IDs in the interval $[0, h-1]$. The lookup table $\mathcal{P}_C$ is a 2D array of size $(h+1) \times n$ and stores offsets into the respective fragments array designating the beginning of a fragment. All fragments are stored consecutively and byte-aligned in one *fragment byte array* per column. Specifically, $\mathcal{P}[t][m]$ stores the offset where fragment $\pi_{C_m}\sigma_{C=t}(R)$ starts in C_m's fragment byte array, where C_m is the $(m+1)$-th column of R. If a value $t \in C$ has no associated values in columns $C_1, ..., C_n$, then all fragments $\pi_*\sigma_{C=t}(R)$ are empty. The size of a fragment is defined implicitly as the difference between two consecutive offsets, which is why the size of the first dimension of $\mathcal{P}$ is $h+1$. For further space savings, offsets are encoded with the minimum number of bytes. In the following, we present various encodings for fragments utilized in this paper.

Fragments Encoding Methods. In a GQ-Fast index $\mathcal{I}_{R.C}$ of relation $R(C, C_1, C_2, ..., C_n)$, all the values associated with $t \in C$ in column C_i are organized as a fragment $\pi_{C_i}\sigma_{C=t}R$. GQ-Fast compresses fragments with different compression methods. GQ-Fast does not have any restrictions on compression methods, as long as fragments can be decompressed without accessing other fragments. It allows a wide range of encoding methods, including those that do not support random access within a fragment. GQ-Fast currently uses the following four methods for encoding single fragments. The extended version provides more details on describing the encoding methods.

- *Uncompressed Array (UA):* An uncompressed array stores the original numerical values in their declared type.
- *Bit-aligned Compressed Array (BCA):* Assume a foreign key attribute points to the IDs of an entity, which range from 0 to $h-1$. Then each foreign key value needs $\lceil \log_2 h \rceil$ bits. Consequently, a fragment $\pi_A\sigma_{F=c}R$ with size n requires $\lceil \frac{n \cdot \lceil \log_2 h \rceil}{8} \rceil$ bytes (including alignment-induced padding).
- *Byte-aligned Compressed Bitmap (BB):* Given an array of values $[v_1, \ldots, v_n]$, the equivalent uncompressed bit vector is a sequence of bits, such that the bits at the positions $v_1, \ldots, v_n$ are 1 and all other bits are 0. GQ-Fast uses the byte-aligned method to compress bit vectors [6]. The first bit of a byte is a flag that declares whether (i) the next seven bits are part of a number that also uses consequent bytes or (ii) the remaining seven bits actually represent the length number by themselves.
- *Huffman-encoded Array (Huffman):* GQ-Fast employs Huffman encoding with an array-based encoding of the Huffman tree [13, 24] to avoid tree traversals (i.e., random access on the heap). This can speed up decoding due to CPU L1/L2 caching effects.

We compared the performance and storage tradeoff of all encoding methods analytically and experimentally. Table 1 summarizes the space needed by each fragment. Assume that each fragment contains N elements, the domain size of the column containing this fragment is D, $E_D = -\sum_{i=1}^{D} p_i \log p_i$ is the entropy of the

Uncompressed Array (UA)	$32 \cdot N \cdot \lceil \log_{2^{32}} D \rceil$
Bit-aligned Compressed Array (BCA)	$8 \cdot \lceil \frac{N \cdot \lceil \log_2 D \rceil}{8} \rceil$
Byte-aligned Compressed Bitmap (BB)	$N \cdot (8 \cdot \lceil \log_{128} \frac{D-N}{N} \rceil)$
Huffman	$8 \cdot \lceil \frac{N \cdot E_D + D}{8} \rceil$

Table 1: Space Analysis of Encoding Methods

column, and p_i is the probability of occurrence of element i[8]. In our experiments, GQ-Fast chooses an optimal encoding for each column with minimal space cost by using the formulas in Table 1, where N is set to be the average fragment size on each column.

Building GQ-Fast Indices. For an entity table $E(ID, M_1, \ldots, M_m)$, GQ-Fast chooses the ID column as the indexed column and creates one index $\mathcal{I}_{E.ID}$. Note that in an entity table, a fragment contains only a single value.

For a relationship table $R(F_1, F_2, M_1, \ldots, M_m)$ with two foreign keys F_1 and F_2, GQ-Fast chooses both F_1 and F_2 as indexed columns, which means GQ-Fast builds two indices $\mathcal{I}_{R.F_1}$ and $\mathcal{I}_{R.F_2}$ according to different indexed columns. The reason is that a relationship table refers to a collection of (potentially undirected) edges in graphs, and it is necessary to provide an efficient way to obtain fragments for both source vertices (in column F_1) and destination vertices (in column F_2). For scenarios where relationship tables have more than two foreign keys, say $a > 2$, to fully index all the foreign key columns (if needed) GQ-Fast builds a indices, which may require a large amount of space.

Incremental Updates. GQ-Fast's compact storage strategy (storing all the fragments of the same attribute in one big fragment array and using offsets to refer to them) can significantly reduce space costs at the expense of incremental updates. To support incremental updates, GQ-Fast could (i) store each fragment independently and (ii) maintain explicit pointers for them. Theoretically, GQ-Fast will then require additional $N(64 - \lceil \log_2 N \rceil)$ bits, where N is the total number of distinct values in the indexed column.

As fragments may be encoded using Huffman encoding, it is challenging to maintain the optimality of Huffman-encoded fragments after massive updates. Dynamic Huffman encoding [22] should be applied, which remains optimal as the weights change.

5. GQ-FAST QUERY PROCESSING

The GQ-Fast Query Processor (Figure 4) transforms a given query into an RQNA expression, which is then transformed into a plan of physical operators (e.g., the plan in Figure 8 corresponds to the RQNA expression in Figure 7(e)), which is then used together with metadata for C++ code generation. This section formally describes RQNA expressions, presents physical operators and key intuitions in the translation of RQNA expressions into plans, and describes how the GQ-Fast code generator translates plans into code, essentially by mapping each physical operator to an efficient code snippet and stitching these snippets together.

5.1 RQNA Expression

To efficiently answer relationship queries, GQ-Fast first translates them into RQNA (Relationship Query Normalized Algebra) expressions (Figure 6). In the simplest case, an RQNA expression is a left-deep series of joins with a selection and aggregation: In Line 4 the RQNA expression starts with a selection $\sigma_c(T \mapsto v)$ of qualifying entities – we call them the *context* entities[9]. Subsequently, the RQNA expression performs a series of left-deep joins (Line 3) that navigate to entities related to the qualifying entities. Optionally, an RQNA expression may group-by the key attribute k (Line 1), followed by multiple aggregations.

In more complex cases, an SQL query (as shown in later examples) may contain nested queries using IN syntax, where IN translates to semijoins (Line 5). Nested queries are themselves relationship queries (Lines 6) without aggregation or the result of an intersection (Lines 7). Figure 7 shows the RQNA expressions for all the queries evaluated in this paper.

$$
\begin{array}{lcll}
RQNA & \Rightarrow & \gamma_{k;f_1(.)\mapsto N_1,\ldots,f_n(.)\mapsto N_{n_\gamma}}\, Join & (1)\\
 & & \quad \text{attributes named } k \text{ are primary or foreign keys} & \\
 & | & Join & (2)\\
Join & \Rightarrow & Join \ltimes_{j.k_1=v.k_2} (\pi_{\bar{A}}(T \mapsto v)) & (3)\\
 & & \quad j \text{ is a variable defined by } Join & \\
 & | & \pi_{\bar{A}}(\sigma_c(T \mapsto v)) & (4)\\
 & | & \pi_{\bar{A}}((T \mapsto v) \ltimes_{v.k_1=x.k_2} Context) & (5)\\
 & & \quad x \text{ is a variable defined by } Context & \\
Context & \Rightarrow & \pi_{v.k} Join & (6)\\
 & | & \pi_{v.k}\sigma_{c_1}(T_1 \mapsto v) \cap \ldots \cap \pi_{v.k}\sigma_{c_n}(T_n \mapsto v) & (7)
\end{array}
$$

Figure 6: Grammar Describing RQNA Expressions

5.2 Physical Operators

This section explains the physical operators' syntax and semantics, neglecting for now the bottom-up pipelined execution aspects.

Fragment-based Join. The operator $\vec{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}L$ receives as input the result of an expression L that produces a column B, generally among others. For each value $b \in B$, the operator uses the index $\mathcal{I}_{R.B'}$ to retrieve (and decompress) the fragments $\pi_{A_i}\sigma_{r.B'=b}\,(R \mapsto r)$ for $i = 1, \ldots, n$. Intuitively, L would be the left operand of a conventional join and $R \mapsto r$ would be the right side. Conceptually, one may think that the fragments are combined into a result table whose schema has the attributes $A_1, \ldots, A_n$ (and also the attributes of L). However, in reality, the decompressed fragments are not combined into rows. In adherence to the late binding technique [1, 2] of column-oriented processing, the ordering of the items in the fragments dictates how they can be combined into tuples. The $\vec{\bowtie}$ operator is useful for executing both selections and joins of the RQNA expressions:

- A projection/join combination $\pi_{attrs(L),r.A_1,\ldots,r.A_n}(L \bowtie_{B=r.B'} R \mapsto r)$ where B is an attribute of L and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $L\ \vec{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}$.
- A projection/selection combination $\pi_{r.A_1,\ldots,r.A_n}\sigma_{r.B'=c}(R \mapsto r)$, where c is a constant and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $\{[B : c]\}\ \vec{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}$. Essentially, GQ-Fast reduces the selection into a join, by considering the left-hand-side argument to be a table with a single tuple and a single attribute B, whose value is c.

Fragment-based Semijoin. The operator $\overrightarrow{\ltimes}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}\ \mathcal{I}_{R.B'}L$ operates similarly to the fragment-based join but returns only attributes from $(R \mapsto r)$ if there is a matching tuple in L. It is introduced in the plan when the RQNA expression has an expression $\pi_{r.A_1,\ldots,r.A_n}((R \mapsto r) \ltimes_{B=r.B'} L)$. The operator maintains a lookup structure for values from the B column of L; for each value $b \in B$, the operator checks the lookup structure to find out whether

[8] We report the lower bound of the space needed by Huffman. The space needed by Huffman is bounded by $[8\lceil \frac{N \cdot E_D + D}{8} \rceil, 8\lceil \frac{N \cdot E_D + N + D}{8} \rceil)$ [29].

[9] The condition may be set to true, setting the context to all entities.

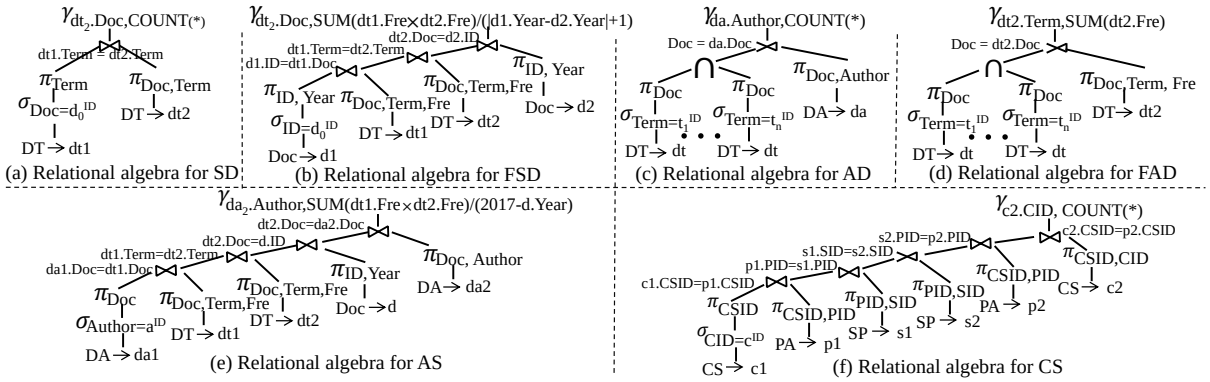

Figure 7: Relational Algebra Expressions for Queries on PubMed/SemMedDB

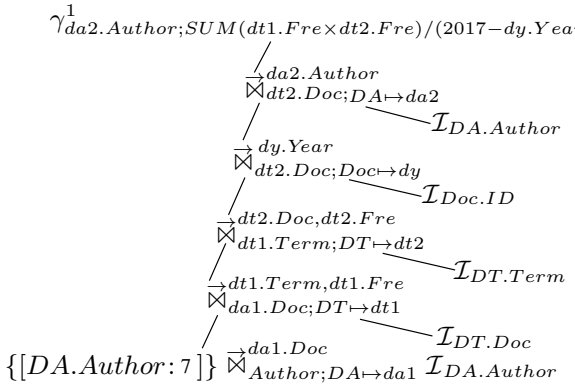

Figure 8: Physical Algebraic Plan for Query AS

that particular value b was already received earlier. If L is relatively large, it is best to use a boolean array, despite the fact that the query needs to initialize all array elements to *false*. Otherwise, a hash set or a tree is preferable.

While joins and semijoins are sufficient, the extended version also describes the occasional replacement of semijoins with a *merge intersection* operator that merges sorted one-attribute relations. As an example, Query AD uses that operator.

Aggregation. The aggregation operator $\gamma^1_{r.D;\alpha(s(A_1,\ldots,A_n))}$ groups its input according to the single group-by attribute $r.D$ and aggregates the results of the scalar function $s(A_1,\ldots,A_n)$ using the associative aggregation function α (e.g., min, max, count, sum). Recall, in relationship queries the single group-by attribute $r.D$ is the foreign key of a relationship table or the ID of an entity. In either case, the range of $r.D$ is the same as the range of the underlying entity ID. Consequently, the γ^1 operator's superscript 1 signifies the assumption that the domain of G is small enough to allow for the allocation of an array, whose size is the domain of $r.D$ and each entry is a number, initialized to zero. Every time a "tuple" from r is processed, this array is updated at $r.D$ accordingly. In addition, an array of booleans registers which values of $r.D$ were actually found. For example, the aggregation $\gamma^1_{Author;\frac{SUM(DT_2.Fre\times DT_1.Fre)}{(2017-DY.Year)}}$ of the Query AS, as shown in Figure 8, initializes an array and a boolean register array with sizes of *domain size of* Author.

5.3 Code Generator

The GQ-Fast code generator (Algorithm 1) has two main components: to-be-emitted source code boxed by dotted lines in the pseudocode; and control commands that determine which code pieces should be emitted. The input of the code generator is (1) the physical plan and (2) GQ-Fast metadata, which specifies the encoding of each fragment. The code generator has two phases: (1) initialize necessary buffers (Lines 3–5); and (2) emit code pieces for each physical operator (Lines 6–40). More precisely, in the first phase, the GQ-Fast code generator initializes an array R to store final aggregation results and several boolean arrays for duplicate-checking in semijoin operations. In the second phase, it emits code for selection operators $\{[B:c]\}$ $\overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}$ $\mathcal{I}_{R.B'}$ (Lines 8–10). The getDecodeFragment macro emits code for (1) retrieving a fragment (Lines 1–3) and (2) calling the corresponding decode macro (Lines 4–11); encoding information is obtained from metadata. For each join L $\overrightarrow{\bowtie}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}$ $\mathcal{I}_{R.B'}$ and semijoin L $\overrightarrow{\rtimes}^{r.A_1,\ldots,r.A_n}_{B;R\mapsto r}$ $\mathcal{I}_{R.B'}$ operator (Lines 11–23), the generator first checks whether the previous operator operates on an entity table and the operated columns are the same. In that case a for-loop can be avoided (Line 14). For a semijoin operator, one more duplicate-checking step should be added (Line 19). The remaining steps (Lines 20-23) of join/semijoin are identical to the selection operator. For each intersection operator $\overrightarrow{\cap}^{\alpha}_{F_1,\ldots,F_m}$, the generator first identifies whether all the fragments are encoded with the same bitmap encoding (metadata). If so ($\alpha = 0$), the generator emits code to perform intersection directly on encoded fragments (Lines 26–30). Otherwise, it emits code to perform intersection on decoded fragments (Lines 32–36). Then, the code generator emits aggregation code pieces (Lines 38–40) for an aggregation operator $\gamma^{1\alpha}_{r.D;\alpha(s(A_1,\ldots,A_n))}$.

Memory Requirements. Query execution requires $4 \cdot |r.D| + \sum_{i=1}^{k} \cdot |r_i.B'|$ bytes, where $|r.D|$ is the domain size of $r.D$ for an aggregation operator, k is the number of semijoin operators, and $|r_i.B'|$ is the domain size of $r_i.B'$ for the i^{th} semijoin operator.

6. EXPERIMENTS

We evaluated GQ-Fast's novelties by running relationship queries on three real-life datasets. In all experiments, the entire data set was located in main memory.

6.1 Environment and Setting

All experiments were done with GQ-Fast 0.1 on a computer with a 4th generation Intel i7-4770 processor (4 × 32 KB L1 data cache, 4 × 256 KB L2 cache, 8 MB shared L3 cache, 4 physical cores,

Algorithm 1: Code Generator

Input: a list of physical operators $\mathbb{O}$ and metadata $\mathbb{M}$;
Output: executable C++ code;
// Initialize arrays in global
Initialize an array R ($|R| = |r.D|$) for $\gamma^1_{r.D;\alpha(s(A_1,\ldots,A_n))}$;
for *each semijoin operator* $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ **do**
  Initialize a boolean array BA ($|BA| = |R.B'|$) with false values;
// Produce codes
for *each physical operator* $o \in \mathbb{O}$ **do**
  if $o = \{[B:c]\} \overset{\rightarrow r.A_1,\ldots,r.A_n}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ **then**
    offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
    for(each column $r.A_i$){
      getDecodedFragment($\mathcal{P}_r, r.A_i, c$);
  else if $o = L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'} || o = L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_B} \mathcal{I}_{R.B'}$ **then**
    //Let o' be the previous operator of o
    if *$o.B = o'.B$ AND $o'.R$ is an entity table* **then**
      $v_B = \mathcal{A}_B[i_B]$;
    else
      for($i_B = 0; i_B < n_B; i_B++$) {
        $v_B = \mathcal{A}_B[i_B]$;
    if $o = L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_B} \mathcal{I}_{R.B'}$ **then**
      if($BA[v_B] = false$) {
    offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[v_B]$;
    for(each column $r.A_i$){
      getDecodedFragment($\mathcal{P}_r, r.A_i, v_B$);
    }
  else if $o = \overset{\rightarrow \alpha}{\cap}_{L_1,\ldots,L_m}$ **then**
    if $\alpha = 0$ **then**
      for(each $L_i = \{[B:c]\} \overset{\vec{\bowtie}^A}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'}$) {
        offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
        fragment $\mathcal{F}_{r.A} = \mathcal{P}_r[column(A)]$;
      }
      $I \leftarrow$Bitwise($\mathcal{F}_{r_1.A},\ldots,\mathcal{F}_{r_m.A}$);
    else
      for(each $L_i = \{[B:c]\} \overset{\vec{\bowtie}^{r.A}}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'}$) {
        offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
        getDecodedFragment($\mathcal{P}_r, r.A, c$);
      }
      $I \leftarrow$Merge($\mathcal{A}_{r_1.A},\ldots,\mathcal{A}_{r_m.A}$);
  else if $o = \gamma^1_{r.D;\alpha(s(A_1,\ldots,A_n))}$ **then**
    for($i_{r.D} = 0; i_{r.D} < n_{r.D}; i_{r.D}++$){
      $R[i_{r.D}] = \alpha(s(A_1,\ldots,A_n))$;
    }
Emit corresponding close braces;

Macro getDecodedFragment($\mathcal{P}_r, r.A, c$)
fragment $\mathcal{F}_{r.A} = \mathcal{P}_r[column(A)]$;
offset-array $next = \mathcal{I}_{R.B'}[c+1]$;
length $l_{r.A} = next[column(A)] - \mathcal{F}_{r.A}$;
if *$\mathcal{F}_A$ is UA encoded* **then**
  decodeUA($\mathcal{F}_{r.A}, l_{r.A}: \mathcal{A}_{r.A}, n_{r.A}$)
if *$\mathcal{F}_A$ is BCA encoded* **then**
  decodeBCA($\mathcal{F}_{r.A}, l_{r.A}: \mathcal{A}_{r.A}, n_{r.A}$)
else if *$\mathcal{F}_A$ is BB encoded* **then**
  decodeBB($\mathcal{F}_{r.A}, l_{r.A}: \mathcal{A}_{r.A}, n_{r.A}$)
else if *$\mathcal{F}_A$ is Huffman encoded* **then**
  decodeHuffman($\mathcal{F}_{r.A}, l_{r.A}: \mathcal{A}_{r.A}, n_{r.A}$)

3.6 GHz), 16 GB RAM, and a Seagate ST2000DM001-1CH1 hard drive, running Ubuntu 14.04.1. Generated C++ code was compiled with g++ 4.8.4, using -O3 optimization.

Dataset. We evaluated all selected DB systems and design choices with three datasets: PubMed-M, PubMed-MS and SemMedDB. Table 2 and Table 3 summarize their data characteristics (see Sections 1 and 2 schemas).

Table name	# rows
DT(Doc,Term,Fre)	207,092,075
DT(Doc,Term,Fre)	901,388,401
DA(Doc,Author)	61,329,130
Document(ID,Year)	23,176,635

Entity ID	Domain Size
Doc(ument)	23,326,299
Term	27,883
Term	259,728
Author	6,301,521

Table	Fragment	Average size	Maximal size	Standard deviation
DT	Doc	7427.18	8192342	197.56
	Term	14.48	667	17.52
DT	Doc	3470.50	8192342	318.72
	Term	63.06	753	39.06
DA	Doc	5.99	5712	21.93
	Author	4.35	3163	8.04
Document	Year	1.00	1	0.00

Table 2: Data Characteristics of PubMed-M and PubMed-MS. PubMed-MS has more terms than PubMed-M, which results in larger size of DT table in PubMed-MS. Gray cells indicate the difference between them.

Table	# rows	Table	# rows
CS	1550482	Concept	1339227
PA	37508726	Sentence	146055876
SP	81929321	Predication	17359895

Table	Fragment	Ave size	Max size	Standard deviation
CS	concept_semtype_id	1.16	5.00	0.39
	concept_id	1.00	1.00	0.00
PA	predication_id	122.00	109532	845.15
	concept_semtype_id	2.15	38	0.53
SP	sentence_id	4.65	125367	112.36
	predication_id	1.61	140	1.07

Table 3: Data Characteristics of the SemMedDB Dataset

Compared Systems. To provide an end-to-end comparison, we compared GQ-Fast with the graph database *Neo4j* 2.3.2[10], the row-oriented database *PostgreSQL* 9.4.0, the cluster-based and column-oriented *Vertica* Analytics Platform, and the in-memory column database *MonetDB*.

To isolate the effect of compiled code from the other contributions of GQ-Fast, we implemented two main-memory column databases serving as main-memory baselines. One is a *plain main-memory column database (PMC)* without optimizations. The other one is a *fully optimized main-memory column database (OMC)*. They both utilize a code generator for executable C++ plans. The logical query plans in PMC and OMC are identical to the ones in GQ-Fast (same RQNA expressions). Both PMC and OMC use the operator-at-a-time execution model as MonetDB [35, 3] does. PMC maintains one copy of each unsorted table, and uses whole column scans when executing each operator. OMC maintains two copies of each table, such that each copy is sorted based on one foreign key column. OMC applies all optimizations that can improve the performance of relationship queries: (1) Applying run-length encoding for sorted columns, improving the lookup performance and reducing space costs; (2) utilizing binary search for sorted columns instead of whole column scan. Implementation details of PMC and OMC can be found in the extended version [25].

[10]Since Neo4j does not support SQL syntax, we translated [25] queries into Cypher, Neo4j's query language.

6.2 Experimental Results

We ran the Queries SD, FSD, AD, FAD and AS on PubMed (PubMed-M and PubMed-MS) and Query CS on SemMedDB (see queries in Section 2). We always chose the encoding with the least space costs, if not stated otherwise; even though a different encoding might perform better in terms of running time. We measured the *warm* running time for queries, i.e., each query was run twice but only measured the second time. The extended version [25] provides more information about the selection of query constants and additional commentary on the results.

We measured the overall runtime performance (Section 6.2.1) and the overall space cost (Section 6.2.2) for each algorithm and database. The results show that GQ-Fast outperforms MonetDB and OMC by 10–10^3 and 7–70 times, respectively, and generally uses less space, due to a combination of the following effects:

- *Compilation*: Using a code generator to generate C++ code.
- *Pipelining*: Adopting a bottom-up pipelined execution strategy.
- *Array-l*: Using dense IDs to maintain an array look-up table instead of a hash table.
- *Array-a*: Using dense IDs to maintain an array to store aggregation results instead of hash table.
- *Compression*: Applying aggressive data compression schemes.

The gap between the speedup of GQ-Fast and OMC over MonetDB reveals the power of compiled code. In order to isolate the effect of the other four optimizations, we implemented variants of GQ-Fast and OMC as summarized in Table 4.

	Compile	Pipeline	Array-l	Array-a	Compress
GQ-Fast	✓	✓	✓	✓	✓
GQ-Fast-UA	✓	✓	✓	✓	✗
GQ-Fast-UA(Bin)	✓	✓	✗	✓	✗
GQ-Fast-UA(Map)	✓	✓	✓	✗	✗
OMC	✓	✗	✗	✗	✓[11]
OMC-denseID	✓	✗	✓	✓	✓[12]

Table 4: Summary of Different Variants of GQ-Fast and OMC

GQ-Fast-UA is GQ-Fast with uncompressed arrays (as encoding). In addition, GQ-Fast-UA(Bin) uses binary search instead of array lookup. Therefore, GQ-Fast-UA(Bin) does not have the dense IDs optimization. GQ-Fast-UA(Map) is like GQ-Fast-UA but uses a hash map instead of an array to store final aggregation results. It does not use the dense IDs optimization. OMC-denseID is like OMC but uses arrays instead of hash maps in both lookup and aggregation, which means OMC-denseID has the same lookup and aggregation data structures as GQ-Fast.

To isolate the effect of each single optimization, we conduced further experiments as described in Section 6.2.3.

- To measure the effect of dense IDs, we compared (i) GQ-Fast-UA with GQ-Fast-UA(Bin) (Table 8), and (ii) GQ-Fast-UA with GQ-Fast-UA(Map) (Table 9).
- To measure the effect of using bottom-up pipelining against materializing intermediate results, we compared GQ-Fast-UA with OMC-denseID (Table 10).
- To measure the effect of applying different compressions, we analyzed the performance and space cost of different compressions in GQ-Fast (Table 11).

[11] OMC uses RLE encoding and dictionary encoding.

[12] OMC-denseID uses RLE encoding and dictionary encoding.

	SD	AD	AS
GQ-Fast on general-graphs	1.440	0.407	47.077
Neo4j on general-graphs	137.6	80.2	50121.9

Table 5: Running Time on General Graphs

Section 6.2.4 provides additional experiments to analyze (1) the effect of parallel processing in GQ-Fast, (2) the time required for building GQ-Fast indices, and (3) the time required for loading indices from disk to memory if the indices are stored on disk.

6.2.1 Overall Runtime Performance

Table 6 reports the average running time of each query for each system, using 8 threads[13]. Overall, GQ-Fast shows superior performance for all queries. We further observed that:

- On average, GQ-Fast outperforms Vertica, MonetDB and OMC by a factor of 100, 170, and 20, respectively (see ratio columns). If GQ-Fast only applies UA compression, it will achieve better performance (running time of Query AS on PubMed-M is 4.45s).
- MonetDB outperforms PMC: Its indexed plans perform better than PMC's compiled code. OMC outperforms MonetDB, since (a) OMC uses code generation and (b) has two copies of each relationship table. For example, OMC uses two copies of the DT table in Query SD. Therefore, each OMC lookup is a binary search on the sorted column (hence essentially tieing the index-based lookups of MonetDB) and lookup results are run-length encoded on the sorted column, hence reducing the size of intermediate results.
- High fanout is favorable to GQ-Fast: The improvement over the competing systems is usually higher in the queries SD, FSD, FAD and AS when they use DT of PubMed-M, compared to queries that use DT of PubMed-MS. Term has a higher fanout in DT of PubMed-M. We conjecture that high fanouts ammortize over larger fragments the fixed costs of the decompression routines, therefore extending GQ-Fast's advantages.

We also conducted experiments to evaluate the performance of GQ-Fast on general graphs. As shown in Table 5, GQ-Fast is still about 100x faster than Neo4j, even after incurring the 10x slowdown (due to lack of knowledge on types), which speaks to the applicability of the GQ-Fast techniques in the case of general graphs.

6.2.2 Overall Space Cost

Table 7 presents the overall space costs. GQ-Fast has the lowest space cost in PubMed-M and PubMed-MS. Interestingly, GQ-Fast also uses much less space than PMC even though PMC stores only one copy of each table while GQ-Fast stores two "copies" (i.e., two indices); this indicates the importance of dense compressions. In SemMedDB, GQ-Fast still uses less space than OMC, but more space than PMC. The reason is the fanout of SemMedDB (averaging at 1.16), which dilutes the effect of fragment compression since fragments are very small and space is spent on padding them to full bytes. Even though PMC uses marginally less space than GQ-Fast in SemMedDB, GQ-Fast is still the best overall choice as it is 760 (i.e., 23.58/0.031) times faster (Table 6).

6.2.3 Effect of Each Optimization

Effect of Dense IDs. The dense IDs assumption allows GQ-Fast to use arrays for semijoins and aggregations instead of other data structures like hash maps.

[13] We applied spinlocks [8] to ensure correctness during concurrent access to shared arrays for the semijoin and aggregate operators.

	Query	Join # tuples	Result # tuples	(a) Neo4j	(b) Postgres	(c) Vertica	(d) MonetDB	(e) PMC	(f) OMC	(g) GQ-Fast	MonetDB / GQ-Fast	OMC / GQ-Fast
PubMed-M	SD	22,401,361	6,409,707	14.7	211.2	5.19	10.8	23.36	2.47	**0.230**	47.0	10.7
	FSD	22,401,361	6,409,707	86.6	567.5	12.32	23.9	33.98	5.93	**0.821**	29.1	7.2
	AD	99,734	57,584	7.4	158.1	2.17	6.2	4.82	0.73	**0.037**	86.5	19.7
	FAD	717,487	5,643	18.2	198.6	3.85	8.5	7.16	0.88	**0.064**	70.3	13.8
	AS	147,273,421	6,393,107	5546.5	29520.5	42.34	4474.8	5832.30	194.77	**5.662**	790.3	34.4
PubMed-MS	SD	136,151,592	12,466,510	61.6	741.2	37.17	49.3	400.24	10.25	**1.068**	46.2	9.6
	FSD	136,151,592	12,466,510	146.8	2148.7	53.48	112.8	1892.30	32.80	**4.376**	25.8	7.5
	AD	85,982	64,765	6.9	112.9	5.60	5.2	19.67	0.67	**0.035**	77.1	19.1
	FAD	1,503,368	9,556	11.1	119.6	15.17	3.5	24.99	0.71	**0.062**	56.5	11.5
	AS	1,391,434,113	9,803,226	9604.8	180164.1	362.35	28918.8	33321.74	3083.30	**54.720**	528.5	56.3
SemMedDB	CS	207,191	5,057	21.0	53.1	10.94	4.7	23.58	2.12	**0.031**	151.6	68.4

Table 6: End-to-end Runtime Performance Tests (in seconds). Numbers in bold are the fastest ones.

	(a) Neo4j	(b) Postgres	(c) Vertica	(d) MonetDB	(e) PMC	(f) OMC	(g) GQ-Fast	MonetDB / GQ-Fast	OMC / GQ-Fast
PubMed-M	34.36	20.92	3.21	3.69	3.09	3.49	**1.47**	2.51	2.37
PubMed-MS	112.15	78.90	11.06	13.27	11.42	11.82	**3.51**	3.78	3.37
SemMedDB	10.39	6.84	2.89	1.23	**0.97**	2.05	1.36	0.90	1.51

Table 7: Space Cost for Each System (in GB). Numbers in bold are the smallest ones.

	Ave # lookups	GQ-Fast-UA(Bin)	GQ-Fast-UA	θ
SD	22	247.94	**177.08**	28.58%
FSD	21748262	1129.72	**435.60**	61.44%
AD	23609	38.67	**30.33**	21.57%
FAD	23609	27.84	**25.95**	6.79%
AS	58589421	7364.92	**4510.11**	38.76%
CS	132975	16.21	**8.62**	46.82%

Table 8: GQ-Fast-UA vs. GQ-Fast-UA(Bin) (in ms). The last column shows the improvement, where $\theta = 1 - \frac{\text{GQ-Fast-UA}}{\text{GQ-Fast-UA(Bin)}}$.

	Ave # results	GQ-Fast-UA(Map)	GQ-Fast-UA	θ
SD	27,443,100	908.95	**177.08**	80.52%
FSD	27,307,529	1342.82	**435.60**	67.56%
AD	200,679	34.84	**30.33**	12.94%
FAD	56,518	31.63	**25.95**	17.96%
AS	20,019,297	7766.83	**4510.11**	41.93%
CS	5,057	10.06	**8.62**	14.31%

Table 9: GQ-Fast-UA vs. GQ-Fast-UA(Map) (in ms). The last column shows the improvement where $\theta = 1 - \frac{\text{GQ-Fast-UA}}{\text{GQ-Fast-UA(Map)}}$.

GQ-Fast-UA vs. GQ-Fast-UA(Bin). We conducted experiments to evaluate the performance of retrieving fragments. Table 8 shows the running time of different queries for GQ-Fast-UA(Bin) and GQ-Fast-UA on PubMed-M and SemMedDB[14]. GQ-Fast-UA outperforms GQ-Fast-UA(Bin) for all queries. For example, GQ-Fast-UA saves around 12% running time over GQ-Fast-UA(Bin) for Query AS. In addition, we also observed that, queries with larger number of lookup requests (FSD, AS and CS) benefit more compared to queries with smaller number of lookup requests, e.g., SD and AD.

GQ-Fast-UA vs. GQ-Fast-UA(Map). We measured the benefit of choosing an array for aggregation in GQ-Fast over a hash map by comparing GQ-Fast-UA with GQ-Fast-UA(Map). As shown in Table 9, GQ-Fast-UA outperforms GQ-Fast-UA(Map) for all queries. GQ-Fast-UA performs better for the queries with a large output (e.g., Query AS; GQ-Fast-UA saves about 33% of running time) compared to queries with smaller output (e.g., Query CS).

[14]We achieve similar improvements in PubMed-MS.

	# fragments	# elements	OMC-denseID	GQ-Fast-UA
A_1	7,484,532	51,730,682	4.12	1.02
A_2	9,287,804	65,687,183	22.15	2.24
A_3	87,467,470	619,809,092	74.90	5.38
A_4	184,219,134	1,305,764,797	171.33	13.76
A_5	585,932,678	4,153,322,719	297.47	49.01

Table 10: Avoiding Intermediate Results

Effect of Pipelining. In this experiment we compared OMC-denseID with GQ-Fast-UA in order to measure the benefit of pipelining over materializing intermediate results. OMC-denseID and GQ-Fast-UA have the same lookup data structure and use an array for final aggregation. Table 10 reports the running time of GQ-Fast-UA and OMC-denseID on five instances of Query AS, where each instance queries for another author ID, A_1–A_5. The number of accessed fragments for those queries varies from around 7M to 585M. As shown, GQ-Fast-UA outperforms OMC-denseID by a factor of 15. As the number of accessed elements increases, the running time of OMC-denseID increases significantly, since OMC-denseID materializes larger intermediate result columns.

Analysis of Different Encoding Methods. We analyzed the performance (compression rate and decompression time) of all encoding methods that are employed by GQ-Fast: uncompressed array (UA), bit-aligned compressed array (BCA), byte-aligned bitmap (BB) and Huffman encoding. Table 11 reports the encoded size of each column in the PubMed-MS dataset. As shown, no single encoding is optimal for all columns. Adopting a suitable compression method can significantly save space. For example, by using BB, the space cost of $\mathrm{dt1.Term}$ reduced from 3660.29 MB to 1431.12 MB. The selection of a suitable compression method is based on our analysis results in Section 4.

We also conducted experiments to evaluate the decompression performance of these encoding methods for two kinds of (synthetic) fragments: fragments on foreign key columns containing only unique values (Table 12) and fragments on measure attributes with many duplicates (Table 13). In the former case, we observed that BB achieves the highest compression (saving 69.25% space) and the highest decompression performance (about 30 times faster than Huffman). Huffman has the worst performance, since the domain

	UA	BCA	BB	Huffman
dt1.Term	3605.55	2033.25	**1376.39**	1565.60
dt1.Fre	901.39	454.12	N/A	**142.46**
dt2.Doc	3605.55	2816.93	**1047.71**	2779.37
dt2.Fre	901.39	450.74	N/A	**134.84**
da1.Doc	245.26	198.75	**187.54**	325.70
da2.Author	245.26	**183.95**	205.10	275.56
dy.Year	57.17	**14.20**	N/A	14.29

Table 11: Size of Encoded Columns (MB). The bold fonts show the minimal space for each column. BB only applies for fragments with unique values, so dt1.Fre and dt2.Fre are not encoded by BB.

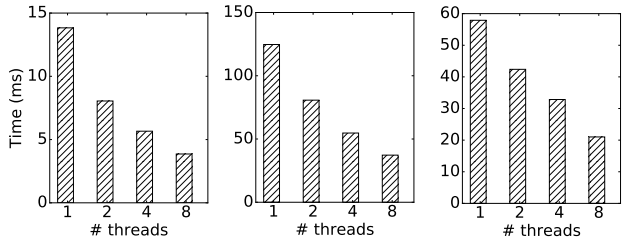

Figure 9: Running Time for Query AS (PubMed-M, PubMed-MS and Query CS (SemMedDB), 1–8 threads

size is large, which requires maintaining a large decoding table (tree) that is too big for CPU L1/L2 caches. In the latter case, we noticed that Huffman achieves the highest compression quality and has decompression performance comparable to BCA, as shown in Table 13. Compared to the results in Table 12, the decompression performance of Huffman improved significantly, because the Huffman table can fit into the L1 cache when the domain size is small (say 100). This result also indicates that Huffman is suitable for measure attributes.

6.2.4 Additional Experiments

Effect of Multiple Threads. We evaluated the effect of multiple threads for the overall performance. Figure 9 shows the running time of selected queries with 1–8 threads. Parallel processing improves performance but does not scale linearly, mostly due to skewed data. For example, the difference between the minimal and maximal number of processed fragments in different threads is around 2 million for Query AS. The skew problem can be solved by employing load-balance algorithms.

Building/Serializing Indices. Table 14 reports the time for building indices in-memory. For GQ-Fast, this process takes a bit more time than for GQ-Fast-UA, since it spends extra time on encoding fragments. The last column shows the ratio, where r is the time required for reading data from disk to memory, which is 361.14s for PubMed-M, 1283.32s for PubMed-MS, and 149.77s for SemMedDB. Table 15 shows the time for (de)serializing indices from/to hard disk. GQ-Fast can do that process faster, since indices are compressed and thus smaller.

7. RELATED WORK

GQ-Fast Indices vs. Database Indices. Database indices (B+ trees, hash indices and bitmaps) are built on top of the original tables, i.e., the database has both tables and indices. In contrast, the entire dataset of a GQ-Fast database is in GQ-Fast indices. This difference has repercussions in query processing. A database index is given a key and returns row IDs. Depending on the system, a row ID may be a tuple pointer, a block pointer or an array index. At any rate, the database then accesses the tuples that are identified by the row IDs and collects the relevant attributes in the original tables. In contrast, the GQ-Fast index is a data-to-data index, which gets rid of row IDs. Given a key and an attribute, it returns directly the attribute values that relate to the particular key.

Comparison of Pipelining Methods. In row-oriented databases, the top-down iterator model [16] reduces the memory footprint of intermediate results. However, the top-down iterator model shows poor performance on modern CPUs due to lack of locality, frequent instruction mispredictions and too many function calls [30]. Therefore, modern column databases choose either to (1) pass blocks of tuples (batch-oriented processing) between operators, reducing the number of function invocations [32, 30], or (2) materialize all intermediate results to eliminate the need of calling an input operator repeatedly, which simplifies operator interaction [27, 19], or (3) choose a middle way by passing large vectors of data and evaluating queries in a vectorized manner on each chunk [42].

However, none of the above techniques reaches the speed of hand-written code [30]. GQ-Fast's code generator compiles physical plans to code to improve performance. While many aspects of GQ-Fast code generation (e.g., function call avoidance) have been employed in previous work employing code generation[15] GQ-Fast produces code that is very close to what a human would do. Crucially, the compiled code utilizes simple for-loops (e.g., see Generated Code in Section 1) that access the elements in a fragment. Tight for-loops create high instruction locality which eliminates the instruction cache-miss problem. Intermediate results are stored in loop variables, such as the $v_{\mathrm{dt1.Term}}$. Such simple loops are amenable to compiler optimizations (e.g., register allocation of loop variables) and CPU out-of-order speculation [19].

Data Cubes vs. Graph Analytics. The fact table of data cubes involves typically $k > 2$ foreign keys. Hence, if we perceive the fact table as a k-ary relationship, we would create k GQ-Fast indices, inducing data redundancy that would eventually surpass the compression advantages. Alternately, we could think of facts as entities, connected to the dimensions via many-to-one relationships. However, once we model a data cube in this way and apply GQ-Fast to it, the benefit of GQ-Fast in performing paths of many-to-many joins is not exhibited anymore. Hence, it becomes apparent that data cube queries and graph queries are significantly different in their SQL OLAP needs and GQ-Fast is tuned towards the latter.

Graph Processing. High-level graph engines allow users to write in SQL or other declarative languages, e.g., Datalog, which is easier to use but orders of magnitude slower [4] than low-level graph engines [17, 31]. GQ-Fast meets the performance of low-level graph engines while supporting a high-level programming interface. It is worth mentioning that, EmptyHeaded [4] has the same design goal as GQ-Fast but they have several differences: (1) GQ-Fast uses an encoded fragment-based data structure, while EmptyHeaded employs a trie data structure; and (2) GQ-Fast focuses on CPU caching effects, while EmptyHeaded focuses on leveraging SIMD (single-instruction multiple-data) to speed up performance.

8. FUTURE WORK

In the future, we will investigate how GQ-Fast can be incorporated in a general SQL processor, where GQ-Fast will execute relationship subqueries and conventional query processing techniques will be used to combine and process the output of GQ-Fast. We will also study the pushing aggregation down optimization in order to further improve the performance.

[15] Among others, generating code is used to speed up data cube queries [30], view maintenance [5] and path-counting queries [28].

	# elements per fragment	# fragments	compression ratio	1 thread	2 threads	4 threads	8 threads
BCA	$100000_{\pm 1000}$	8000	76.23%	1535.506	864.594	450.890	378.227
BB	$100000_{\pm 1000}$	8000	31.75%	1501.806	835.154	428.818	371.442
Huffman	$100000_{\pm 1000}$	8000	73.08%	52198.713	29688.662	14934.780	7925.446

Table 12: Space Cost and Decompression Time for BCA, BB, and Huffman. Domain size is 1 billion, data follows Zipf distribution with factor $s = 1.5$. Fragments only contain unique values, which simulates fragments in foreign-key columns.

	# elements per fragment	# fragments	compression ratio	1 thread	2 threads	4 threads	8 threads
BCA	100	8000000	21.88%	1581.167	801.313	410.039	348.422
	10000000	80	21.88%	1286.579	652.020	333.645	283.507
Huffman	100	8000000	12.28%	5055.162	2543.277	1280.826	668.050
	10000000	80	11.39%	4374.838	2201.003	1108.453	578.143

Table 13: Space Cost and Decompression Time for BCA and Huffman. Domain size is 100, data follows Zipf distribution with factor $s = 1.5$. Fragments contain duplicates, which simulates fragments in measure attributes.

	(1) GQ-Fast-UA	(2) GQ-Fast	$\frac{(1)+r}{(2)+r}$
PubMed-M	152.52	153.89	73.74%
PubMed-MS	540.33	561.02	72.54%
SemMedDB	74.68	103.30	66.15%

Table 14: Running Time for Building Indices (seconds)

	Serializing index time		Deserializing index time	
	GQ-Fast-UA	GQ-Fast	GQ-Fast-UA	GQ-Fast
PubMed-M	139.52	60.50	152.52	70.46
PubMed-MS	495.10	163.92	540.33	187.17
SemMedDB	63.36	49.96	74.68	58.77

Table 15: Time for (De)serializing Indices (seconds)

9. REFERENCES

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[4] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, 2016.

[5] Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[6] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC*, page 476, 1995.

[7] S. Bordoloi and B. Kalita. Designing graph database models from existing relational databases. *IJCA*, 74(1), 2013.

[8] J. Catozzi and S. Rabinovici. Operating system extensions for the teradata parallel VLDB. In *VLDB*, pages 679–682, 2001.

[9] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[10] C. Chen, X. Yan, F. Zhu, J. Han, and S. Y. Philip. Graph OLAP: a multi-dimensional framework for graph data analysis. *KAIS*, 21(1):41–63, 2009.

[11] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[12] P. Chen. Entity-relationship modeling: historical events, future trends, and lessons learned. In *Software pioneers*, pages 296–310. Springer, 2002.

[13] K. Chung and J. Wu. Level-compressed huffman decoding. *TCOM*, 47(10):1455–1457, 1999.

[14] N. Colossi, W. Malloy, and B. Reinwald. Relational extensions for OLAP. *IBM Systems Journal*, 41(4):714, 2002.

[15] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[16] G. Graefe. Query evaluation techniques for large databases. *CSUR*, 25(2):73–169, 1993.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ISCA*, volume 40, pages 349–362. ACM, 2012.

[18] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378. IEEE, 2003.

[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *DEBU*, 35(1):40–45, 2012.

[20] H. Kilicoglu, M. Fiszman, A. Rodriguez, D. Shin, A. Ripple, and T. C. Rindflesch. Semantic MEDLINE: a web application for managing the results of PubMed searches. In *SMBM*, volume 2008, pages 69–76, 2008.

[21] H. Kilicoglu, D. Shin, M. Fiszman, G. Rosemblat, and T. C. Rindflesch. SemMedDB: a PubMed-scale repository of biomedical semantic predications. *Bioinformatics*, 28(23):3158–3160, 2012.

[22] D. E. Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[23] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *VLDB*, 5(12):1790–1801, 2012.

[24] J. Li, H. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *PVLDB*, 9(14):1647–1658, 2016.

[25] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory SQL analytics on relationships between entities. *CoRR*, abs/1602.00033, 2016.

[26] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-SQL: Fast query processing via graph exploration. *PVLDB*, 9(12), 2016.

[27] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[28] B. Myers, J. Hyrkas, D. Halperin, and B. Howe. Compiled plans for in-memory path-counting queries. In *IMDM@VLDB*, pages 28–43. 2015.

[29] G. Navarro and N. Brisaboa. New bounds on D-ary optimal codes. *Information Processing Letters*, 96(5):178–184, 2005.

[30] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[31] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471. ACM, 2013.

[32] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[33] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[34] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-j. P. Hsu, and K. Wang. An overview of Microsoft Academic Service (MAS) and applications. In *WWW*, pages 243–246. ACM, 2015.

[35] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[36] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[37] D. W. Wardani and J. Kiing. Semantic mapping relational to graph model. In *IC3INA*, pages 160–165. IEEE, 2014.

[38] F. Xia, Y. Li, C. Yu, H. Ma, and W. Qian. Bsma: A benchmark for analytical queries over social media data. *VLDB*, 7(13):1573–1576, 2014.

[39] X. Xie. Potential friend recommendation in online social network. In *GreenCom*, pages 831–835. IEEE, 2010.

[40] Z. Xu, S. Zhang, and Y. Dong. Mapping between relational database schema and OWL ontology for deep annotation. In *WI*, pages 548–552. IEEE, 2006.

[41] S. Zhou. Exposing relational database as RDF. In *IIS*, volume 2, pages 237–240. IEEE, 2010.

[42] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 – a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.