Diploma Thesis Exposé

An Implementation Of The Annis System On Top Of A Column-Oriented Main-Memory Database

Viktor Rosenfeld*
Supervisor: Prof. Dr. Ulf Leser

August 21st, 2011

1 Background

1.1 Annis

Annis is a web-based search engine and visualization tool for multi-modal, multi-layer linguistic text corpora [ZRLC09]. Annotations are taken from various tools and merged into a unified corpus with support for conflicting annotations over the same primary data. It is implemented on top of the relational database managements system (RDBMS) PostgreSQL, using a pre/post-order scheme to store the nodes of an ordered directed acyclic graph (ODAG) [Ros11]. This graph encoding is inspired by the XPath Accelerator, originally developed to store XML documents in a RDBMS [GKT04]. However, the Annis data model differs notably from the XML document model:

- Each node can have multiple parents and therefore many pre- and post-order values. These are stored in a separate table.
- A node in the Annis data model represents more information than a node in a XML document tree, such as its position in a linear text and key-value annotations.
- Unlike the edges in a XML document tree, edges in the Annis data model are typed and carry semantic meaning and may also be annotated with key-value pairs.

*rosenfel@informatik.hu-berlin.de

Thus, the database schema used by Annis is more complicated than the schema used by the XPath Accelerator. The latter stores the tree structure of a XML document in a facts table with five attributes and the textual content of XML nodes in four additional binary tables (one each for text, attribute, comment, and processing instruction nodes) which reference the facts table primary key (the pre-order value of the node) in their first column. In total, the XPath Accelerator schema requires nine attributes to completely store a XML document. In contrast, Annis requires six tables with 20 attributes (excluding surrogate primary/foreign keys) to store the annotation graph over a single text, all of which may be referenced in a typical Annis query.

The main optimization strategy in [Ros11] was to denormalize most of the Annis database schema into a single facts table and build operator-specific combined and partial B-Tree indices which allow us to answer both a node look-up and a node relation predicate in a single index look-up, minimizing the data that has to be read from disk. While this strategy was generally successful and allows us to evaluate many queries interactively on modern consumer hardware, it has significant drawbacks:

- The large number of indices increase the space required to store a corpus by an order of magnitude. For example, PostgreSQL requires 428 MB to store the TIGER corpus using the normalized schema without indices. However, the denormalized facts table and the indices built on top of it require more than 5 GB on disk.
- Annis queries with a low selectivity, especially those dominated by unbounded regular expression searches, are slow because they generate a lot of disk I/O by sequentially scanning the facts table.
- PostgreSQL does not support index-only plans and has to touch the facts table for each selected node. Disk I/O is therefore directly dependent on the width of the facts table.

1.2 Column-oriented databases

Traditional databases employ a n-ary storage model (NSM), i.e. they store the values of all attributes of a tuple together. Using a table-based metaphor for a relation, traditional databases store individual rows. In contrast, column-oriented databases, or column-stores, implement a decomposed storage model (DSM), i.e. they store the values of the same attribute of all tuples together. A relation with n attributes is vertically partitioned into n binary attribute relations and the attribute values of a tuple are linked by surrogate keys.

The advantages of the DSM were first described in [CK85]: queries that only reference some of the attributes of a schema can be evaluated faster on the DSM than on the NSM. Because a column-store can load only the attributes that are required to evaluate a query, it reduces disk I/O and makes better use of the buffer cache. More joins are required to evaluate a query on the DSM than on the NSM, but because they operate on thin tables each join is faster.

These theoretical advantages have been realized in a number of applications, including Data Mining [BRK98], Data Warehouses [SBÇ+07], and Scientific Computing [INGK07]. In these fields column-stores can achieve an advantage over traditional RDBMS by an order of magnitude or more. In [AMH08] the authors simulated column-oriented processing in a traditional row-oriented RDBMS, by using index-only plans or artificially partitioning the schema into attribute tables.[1] This did not yield performance gains over a traditional relational schema due to high tuple overhead and relatively slow join performance in traditional RDBMS. Furthermore, denormalization – a typical optimization method which we also used in the original Annis implementation – only yields performance gains for highly compressible data if used in column-stores.

[1] A third technique – specialized materialized views – is not applicable to Annis because the query workload is not known in advance.

1.3 Main-memory databases

As the size of available main memory increases, it has become possible to keep entire databases or at least the hot set of very large databases resident in main memory. Thus, the overriding optimization criteria for traditional RDBMS – the reduction of disk I/O – has become less important and other optimization criteria have become relevant. Moreover, main memory is directly accessible by the processor as opposed to the block-based structure of disk memory. These differences impact all components of a RDBMS [GMS92, MKB09].

Although main memory is randomly accessible, from the point of view of a modern processor not all of it is accessible at the same speed: access to memory regions that have not yet been loaded into the CPU cache hierarchy will incur a cache miss penalty during which the processor effectively stalls if it cannot find work to do in parallel. Indeed, many commercial RDBMS spend much of the query execution time on main memory and other resource stalls [ADHW99]. To achieve optimal performance on modern processors it is necessary to improve the cache behavior of query processing algorithms [SKN94, BMK99], indexing techniques [RR00, CGM01] and storage layout [ADHS01] as well as keep the functional units of modern super-scalar processors occupied at all times [BZN05].

2 Motivation

We believe that Annis is a good candidate to be sped up by the use of a column-store.

Annis accesses the database mostly in a read-only fashion: once a corpus is imported it is no longer modified. It uses a snowflake schema that is common for analytical database workloads.[2] A typical Annis query only puts constraints on some of the facts table attributes. If a query actually does require every attribute of the facts table for evaluation, it will do so by joining the facts table many times with itself. Thus there exists a strong potential for the reduction of unnecessary disk I/O by only accessing the attributes that are required to satisfy a query.

However, Annis does not fit the OLAP model perfectly. Except for trivial queries, Annis joins the facts table with itself many times over. The very common $ANNOTATE$ query additionally projects every attribute of the facts table. In other words, for a small amount of tuples every attribute will have to be accessed, a use case for which column-stores are ill-suited. However, $ANNOTATE$ queries have a very high selectivity because their results are paginated in the setting in which Annis queries are most often used, i.e. the Annis web client fetches the results in blocks of 10 to 50 matches per query.[3] Nevertheless, the performance of a column-store degrades with the number of projected attributes and a traditional RDBMS may be at an advantage at this stage of query processing [HLAM06].

3 Aims

The objective of this work will be a port of the Annis system to a column-oriented main-memory database. The port should be complete enough to support the core functionality of the Annis web frontend, i.e. $COUNT$ and $ANNOTATE$ queries.

Once the port is completed we will compare it to the current Annis implementation. Specifically, we want to answer the following questions:

[2] A key difference is the size of the "dimension" tables containing node and edge annotations: these can be as large or larger than the "fact" tables storing nodes and edges. This appears to be an oversight in the original implementation. The annotation tables are highly redundant and could be substantially reduced in size if the primary/foreign key relationship is inverted.

[3] The size of the result set is not only dependent on the number of requested matches, but also on the user-specified query context as well as the structure of the queried corpus. The selectivity of typical ANNOTATE queries lies between 0.1% and 2% of the facts table.

- Does the Annis system benefit from an implementation on top of a column-oriented database? Is it able to evaluate a typical workload faster or with fewer space requirements?
- Is such an implementation able to work directly on the normalized source tables, avoiding the large materialized facts table and the many combined indices on top of it?
- How do different materialization strategies influence the performance and space requirements of the column-oriented backend? For example, is it beneficial to join the node and edge tables into a facts table to which the annotation tables are linked by primary/foreign key relationships?[4]
- How does the port react to an increase in the complexity of the schema?

4 Approach

We first have to identify the target of the port. In the last decade there has been much research on column-stores and many commercial systems have become available [ABH09]. However, the Annis project requires a software stack consisting entirely of free software. We have identified the following four open-source column-stores:

- *MonetDB*[5], which pioneered the research into column-stores in the late 1990s [BK99, BZN05],
- *C-Store*[6], another research column-store, which has been commercialized as the Vertica column-store and is no longer being developed [SAB+05],
- *Infobright Community Edition*[7], a storage engine for MySQL, and
- *LucidDB*[8], a RDBMS geared towards data warehousing written in Java.

A comparison of these four column-stores is outside the scope of this work. We will focus on MonetDB because it is being actively developed and backed by a strong research community.

The current Annis implementation uses little PostgreSQL-specific functionality and MonetDB follows the SQL 2003 standard. Unfortunately, MonetDB does not implement the standard completely. One of the features missing from MonetDB is support for regular expression predicates. These are used by Annis to implement regular expression text and annotation searches. A limited workaround is possible: regular expressions that encode a fixed number of alternatives can be rewritten using Annis query alternatives. Regular expressions that match an unspecified sequence of characters (.*) can be simulated using a LIKE predicate. Other regular expressions can possibly be implemented with a stored procedure or may have to be rejected altogether. Another approach is to add regular expression matching to MonetDB. We will investigate this possibility, but expect it to be outside the scope of this work.

With the aforementioned exception, we anticipate the port to be fairly straightforward. Since the current Annis implementation already supports the mapping of the normalized source tables to arbitrary materialized views, the study of the effects of different materialization schemes should also require few changes to the Annis code base. The main focus will be column-store-specific optimizations, such as the removal of the redundancy contained in the annotation tables and various compression and clustering techniques.

To study the influence of the schema complexity we want to add support for multiple precedence layers (see below) to the Annis system and measure its influence on the system.

[4] Queries accessing the edge tables experienced a major slowdown on the normalized schema in the original Annis implementation.
[5] http://www.monetdb.org/
[6] http://db.csail.mit.edu/projects/cstore/
[7] http://www.infobright.org/
[8] http://www.luciddb.org/

5 Multiple precedence layers

Currently Annis supports only one precedence layer which is based on the explicit tokenization of the primary text. For some corpora, multiple precedence layers would be desirable.

Figure 1 shows a sentence fragment from the FALKO corpus which contains essays by non-native German speakers. The *tok* layer at the bottom shows a grammatically incorrect sentence fragment as it was originally written in the essay. The *ZH2* layer at the top shows a possible correction and the *ZH2Diff* layer below the type of correction. To model insertions we currently have to insert empty tokens in the *tok* layer which produces incorrect results for precedence operations. For example, the query `"genau" & "was" & #1 . #2` will not find the depicted sentence fragment. A special *word* annotation layer exists as a workaround, i.e. the query could be reformulated as `word="genau" & word="was" & #1 . #2`.

Multiple precedence layers could be implemented by extracting the attributes which model the token order (*token-index*, *left-token* and *right-token*) from the *node* relation and storing them in an additional *precedence* relation which is partitioned by a *type* attribute. The precedence operator is also extended with a *type* component so that it only selects the appropriate entries of the *precedence* relation. In Figure 1, the *token-index* attribute for two hypothetical precedence layers is shown, one for the original text (type *orig*) and one for the corrected version (type *zh2*). One could then use the following query to ask for inserted corrections: `"genau" & "was" & tok & #1 .orig #2 & #1 .zh2* #3 & #3 .zh2* #2`.

This approach is similar to the implementation of multiple, conflicting graphs over the same nodes. The same method could be used to implement hierarchical precedence layers such as sentences or subtokens. A sentence precedence layer could be used as the context for *ANNOTATE* queries.

	1	2	3	4		5	6	7	8	9	10	11	12	*zh2*
ZH2	dann	werden	alle	Leute		genau	das	machen	,	was	sie	wollen	.	
ZH2Diff					MOVS		INS	MOVT	INS					
word	dann	werden	alle	Leute	machen	genau				was	sie	wollen	.	
tok	dann	werden	alle	Leute	machen	genau				was	sie	wollen	.	
	1	2	3	4	5	6				7	8	9	10	*orig*

Figure 1: Hypothetical multiple precedence layer example from the FALKO corpus. The original text written by a non-native German speaker (layer *tok*) has been corrected by inserting two new words and moving one word to another position (layer *ZH2* and *ZH2Diff*).

References

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented Database Systems. In *Proceedings of the VLDB Endowment*, August 2009.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 1999.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[BRK98] Peter A. Boncz, Tim Rühl, and Fred Kwakkel. The Drill Down Benchmark. In *Proceedings of the 24th International Conference on Very Large Data Bases*, 1998.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining Query Execution. In *Proceedings of the 2007 Conference on Innovative Data Systems Research*, 2005.

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 2001.

[CK85] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, 1985.

[GKT04] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems*, 2004.

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006.

[INGK07] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, 2007.

[MKB09] S. Manegold, M. L. Kersten, and P. A. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *Proceedings of the International Conference on Very Large Data Bases*, 2009.

[Ros11] Viktor Rosenfeld. An Implementation Of The Annis 2 Query Language. Studienarbeit, Humboldt-Universität zu Berlin. http://www.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/studienDiplomArbeiten/finished/2010/rosenfeld_studienarbeit.pdf, 2011.

[RR00] Jun Rao and Kenneth A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005.

[SBÇ+07] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? – Part 2: Benchmarking Results. In *Proceedings of the 2007 Conference on Innovative Data Systems Research*, 2007.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[ZRLC09] Amier Zeldes, Julia Ritz, Anke Lüdeling, and Christian Chiarcos. Annis: A search tool for multi-layer annotated corpora. In *Proceedings of Corpus Linguistics*, 2009.