Compressed, Memory Resident, Databases

W. P. Cockshott J.Gilchrist D. McGregor P. Murray
J. Wilson

August 26, 1996

Department of Computer Science
University Of Strathclyde
26 Richmond Street
Glasgow
G1 1XH
Scotland

Keywords : Persistence, Flash Memory, Database, Compression, Relational

Abstract

The paper argues that recent developments in solid-state store allow a new implementation paradigm for databases. This involves replacing slow rotating storage with all-semiconductor data stores. The relatively higher costs of semiconductor store make data compression advantageous. It is argued that relational databases are well suited to this. Empirical measures of the potential performance gains from compression in commercial databases are presented.

1 A New Database Architectural Paradigm

For much of the last thirty years, the architecture of large computer- based information systems has been dominated by the high cost of available high-speed main memory technology, and the long latency of large backing storage devices. Indeed the data structures and associated access methods developed for database and information retrieval have been dominated by these considerations. High-speed RAM storage was used only for volatile or ephemeral data - the 'authoritative' representation of the data was deemed to be that on the non-volatile backing store. This was only partly due to the volatile storage characteristics of dynamic RAM (magnetic core storage was non-volatile), while reliable battery-backed RAM was available in other application areas.

Thus large volume databases have been backing-store based, data existing only temporarily in RAM for access and manipulation. As indicated above, this has historically been a cost/performance issue, but the contemporaneous quest for consistency and reliability led to the handling of these, too, crystallizing along the lines of the main-store/backing-store discontinuity.

In a conventional disk-based database architecture, access to data is normally to specific data items (in typical commercial transaction processing), or to small subsets of the data (in Relational queries). In general the complete dataset is only handled directly for purposes of checkpoint or 'health-check'. Data is normally stored in an 'indexed' or 'addressable' form such that required information can be accessed with low complexity, and in particular with few backingstore transfers.

This paper's thesis is that the old paradigm is now far from optimal, and that a radical re-design may lead to much higher performance with lower costs. The essential factors underlying the new paradigm are:

- The absolute costs of both RAM and other VLSI components such as processors, continue to drop steeply.
- The relative cost/performance of system components have not changed. While the cost differential between disk storage, and electronic storage is narrowing, the mis-match between processing and backing-store performance is widening. For a variety of reasons mainly concerned with improving the match between processor and storage performance, it is becoming common for designers to provide much larger RAM.
- The need for higher performance database servers to support corporate information systems. Increased transaction rates are one aspect; increased use of systems for strategic decision making, requiring access to large sets of data, is another.
- The reduction of cost and the increase in available processing power, not only in single processors, but in distributed and parallel configurations.

The ethos of the new paradigm is as follows:

- High-performance databases should now be held in direct-mapped storage (e.g. RAM). The cost of storage is minimized by data compression techniques, which can in general reduce the cost of storing data to the same order (or less) than that for a traditional backing-store based design.
- A 'change of viewpoint' in that the main-store representation of an entity is now regarded as the 'authorized version' - the backing-store version as the 'copy'. This change eliminates frequently-incurred access delays when initial transaction data is first read from backing storage, and also write-through delays on completing updates. It also places stringent new reliability requirements on the 'main store' technology to be used in this new role.

The improved access of processors to data, and the greater processing power now available, more than compensates for the increased processing required (even when handling highly- compressed data). In general the time spent in compression/decompression is less than the time which would be lost in waiting for the corresponding I/O. Even more important, is that under the old paradigm it is very difficult for the designer to reduce the time spent on I/O, even if all I/O is effectively to RAM. Means are at hand, and developments are underway which will reduce compression/decompression delays and cost.

Data compression produces an apparent improvement in disk transfer rates. It also reduces disk storage requirements and costs, and the cost of any ultimate archive.

In general under the old paradigm the designer had to trade space, and particularly backing-store space, for access speed and reduced processing overheads. In the new paradigm the designer trades processing/RAM performance to achieve reduced storage volume, with lower I/O overheads. A key requirement is that the new architecture must not result in a lower data reliability as a consequence of the changed placement of the authorized version of the data in high-speed memory. The systems design and the technology to achieve this is the major topic of this paper.

Research into supporting large directly-addressable and non-volatile (i.e. persistent) data-stores is largely derived from the early work of Kilburn [8] on virtual memory. This was concerned primarily with making disk look like RAM, and to do this introduced the notion of the single level store. The single level store was further developed by IBM [13, 7, 9] in the System/38 to support relational databases. More recently a number of research projects in Australia have focused on hardware support for persistence with a view to constructing more reliable operating systems, [1, 10, 11]. These and other similar systems [12], have relied upon memory management technology to map areas of the disk into volatile semiconductor store.

In the past, main memory was at least two orders of magnitude more expensive than contemporary disk storage, much too large a gap to be crossed, despite the attractive feature of its much greater bandwidth (typically 5 orders of magnitude in access bandwidth). However, the reduction in cost and consequent recent availability of large RAM storage has begun to alter the architectural scene. Historically, over the last 10 years the capacity of solid-state memory devices has quadrupled every three years. There is an expectation [11] that 1 GByte DRAM devices will be available within 10 years.

As an interim method of improving disk access performance, many systems now use large-volume disk caches. This produces immediate benefit relative to mechanical storage, but one which fails to exploit the true performance potential of the solid-state memory.

Semiconductor memory prices depend primarily upon their costs of production and secondarily upon current market conditions. When a competitive market exists the prices of a memory technology tend to be determined by the amount of labour required to manufacture the basic components. However, when a new technology is introduced and for reasons of patents or technical difficulties only one or two firms are capable of producing it, then a monopoly price will exist that may be considerably above what would be justified by production costs. In the long term, however, prices expressed in terms of $ per bit, show an exponential decline. This is an expression of the fact that the labour required to build a chip is relatively constant from one generation to the next, whilst between generations there is generally a fourfold rise in storage capacity.

Figure 1 shows the negative exponential evolution of storage prices for disks and dynamic RAMs over the last 17 years. The source of the data was adverts in the May issues of BYTE since 1978. For DRAMs, two prices are usually shown, being those of historically overlapping chip generations. Prices are shown on a logarithmic scale in historic cost terms. Since no allowance is made for inflation, the graph underestimates the fall of prices in constant purchasing power dollars.

An interesting feature is that the rate of decline of disk prices is faster than that for semi-conductors. This may reflect the fact that the cost per bit of a disk declines for two reasons :

1. Improvements in medium coercitivity and reduced head clearance, allow an increas in bits cm^{-2}. This is analogous to the decline in feature sizes in semiconductors.

2. The electronics and mechanism of the drive assembly become cheaper for generation to generation. The electronics become more integrated and the mechanics simpler. This is reflected in an absolute decline in the price of entry level disks. This process has no analogy in semi-conductors.

The increasing economic viability of main memory databases has prompted a considerable amount of research effort to be devoted to the investigation of main memory database systems [2], [14].

prices.eps

Figure 1: Price trends for disk and DRAM store.

A typical approach to data representation in such systems is to characterise domain values as a sequence of strings. Tuples are represented as a set of fixed length pointers to corresponding domain values [5]. Each domain is associated with an inverted list of object identifiers which identify the tuples containing the domain values. References can therefore be made in both directions between tuples and domains.

This approach provides significant compression but leaves each domain value represented by a fixed length pointer which would typically be a machine word in length. An alternative strategy is to store a fixed length reference number for each domain value instead of a pointer to the value [3]. The benefit of this approach is that in static file structures, careful choice of the reference number will provide optimal packing of tuples. Goldstein and Strnad also identify the need to vary compression strategy to match relations which contain different types of data.

An optimal strategy for generating lexical tokens is describe by Wang and Lavington [16]. This approach provides for the possibility of representing domain values in dynamic files by a series of minimal length reference numbers and gives the potential for the most efficient packing of compressed tuples.

When compared to DRAM, magnetic disks have had two great advantages: they were cheap and they were non-volatile. To set against this one has to put the much higher speed of RAM. Now, with the invention of flash memory, disks retain only a cost advantage over the offerings of the semiconductor manufacturers. DRAM memory currently has a price of around £20 a megabyte, which is roughly where disk prices were in 1988. If we assume that this 7 year lead by the disk manufacturers will continue, or even grow for some time into the future, disks will remain the medium of choice for the very largest databases. But the corollary of this is that an application that was mounted on disks 5 years earlier could, for the same cost, now be implemented in solid-state store, with, one hopes, considerable performance gains.

The random seek time for a disk is of the order of 10ms, whereas random access to DRAM memory is under 100ns. This is a difference of 5 orders of magnitude. Once a location on disk has been found, the difference in bandwidths is more modest, perhaps 132 megabytes per second over a PCI bus against about 10 megabytes per second on an SCSI disk drive.

The remaining problem is that of non-volatility. In many applications, where one is dealing with query only systems, this is not an issue, but where it is, there are obvious technical solutions. Battery back-up systems for memories are already extensively used in applications where continuous operations are essential - such as in the processors controling telephone exchanges. But it is not even necessary to maintain the entire database in non-volatile memory. It is sufficient to have a relatively modest amount of battery backed memory to hold part of a transaction log file, which can be periodically flushed to disk. The important fact is to ensure that there is no disk latency on individual transactions.

2 Implications for Databases

Although there has been considerable research into memory resident databases [2][14] most current DBMSs are built around disk resident data. They require external data structures to alleviate the disk I/O bottleneck and are optimized for speed rather than disk space occupancy. For instance Oracle Version 6 uses table spaces which are fixed sized OS files. Within these, table spaces are allocated to tables and indices in fixed size extents, and the table extents are maintained partially empty to permit row insertions.

Given that solid state memory is relatively more expensive, we would wish to avoid any redundant use of space. It thus seems worthwhile to investigate the effect of using compression techniques to achieve a maximal entropy and minimal volume representation.

Since compression techniques invariably introduce an element of indirection into data access, looking up dictionaries for example, the performance costs on disk are relatively high. Disk databases have to be optimized to reduce the number of distinct disk seeks. This penalizes any data structure that relies upon following pointers. In random access memory, the same constraint no longer applies. Following a chain of pointers becomes cheap.

'Tokenization' (in which lexical analysis is applied to the data prior to storage), has been applied in some database and information retrieval systems, but not others. The balance is between the greater compactness of storage, with the greater efficiency of some storage operations, versus the greater software complexity and processing overheads of detokenizing, particularly if the symbol table is backing-store resident.

More intense forms of data compression have been applied; the Peterlee Relational Test Vehicle PRTV - an early relational database prototype [15] applied two-dimensional compression, by sorting similar records into adjacent locations and storing only a flag bit per field, with the differences. However such methods result in variable length tuples, destroying the simple addressability of fixed-length tuples. At that time, compression/decompression then became the dominant element in processing (partially alleviated by microcoding the low-level compression/decompression operations). The compression ratio was however excellent, but the CPU overhead of compression was too great for the technology of 1976.

It should be possible to exploit redundancy inherent in the tabular structure of a RDBMS and also to eliminate the space overheads that arise when working with disk files. The sort of stream oriented compression techniques like Huffman or LZW coding[17] that are often applied to disk applications are not relevant here because:

- we want relational operations to work on the compressed data,
- we want the tuples to be of fixed length to aid column indexing,
- we want random access to the rows of the table.

Instead we propose a compression mechanism that translates the values in each column of a table into short integers. These act as indices into a dictionary that is unique to the domain of the column. The bit-fields representing column values are then concatenated to form the rows.

Initial experiments on a commercial Paradox 4.5 (for DOS) database of 5 tables yielded promising results.

A database that occupied 2.5MB in Paradox was compressed down to 329KB. In section 2.1 a detailed account of the compression experiment is presented.

2.1 First order compression

Consider the simple relational database shown in table 1. It has a single relation CHIPSPEC that describes the characteristics of a number of chips.

If this is stored as a file of records, we have to make each field of these records big enough to hold the largest value that any of the records stores in that field. Since, moreover, the database designer is not likely to know exactly how large the

Table 1: CHIPSPEC Database Contents

PART	**PINS**	**TECHNOLOGY**	**PROGRAMMABLE**
ep900	40	CMOS	Yes
ep310	20	CMOS	Yes
DMPAL16L8	20	TTL	Yes
DMPAL10H8	20	TTL	Yes
MPM5032	28	CMOS	No
CY2901	40	CMOS	No
CY2909	28	CMOS	No

individual values may be, the tendency will be to err on the side of caution and make the fields longer than is strictly necessary. In this instance a designer might specify the following field widths in bytes.

PART	10
PINS	3
TECHNOLOGY	4
PROGRAMMABLE	3
Total	20

With each tuple occupying 20 bytes the space occupied by the relation would be 140 bytes. There are certain simple tricks that could be used by the database designer to reduce this.

The PROGRAMMABLE field could be reduced to a single character, `y` or `n`. The PINS field could be expressed as a short integer, reducing its width to 2 bytes. But the length of each tuple would only be reduced to 17 bytes by such means.

First order compression involves reducing each field to an integer containing just sufficient bits to encode all the values that occur within the domain of that field in the database. Since there are seven distinct part names, the parts column of the relation could be represented as a list of 3 bit numbers. Similarly, since there are only 3 distinct values for the number of pins, these could be represented as 2 bit numbers.

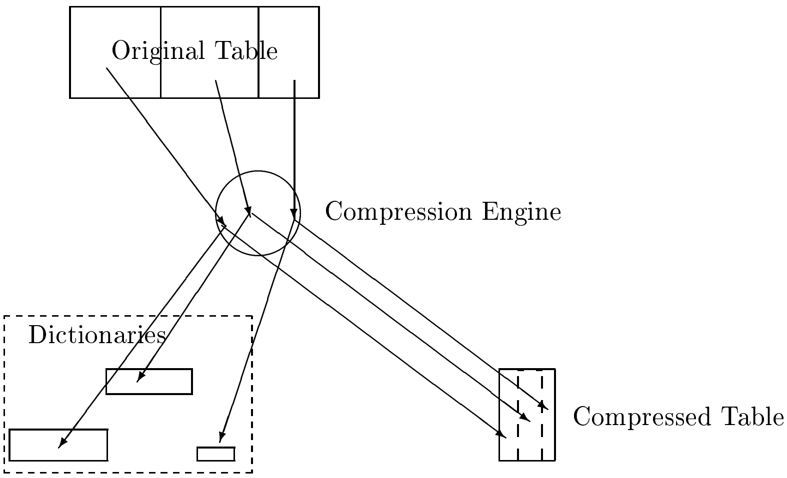
Once similar compression has been applied to the other fields, the resulting tuples are found to be only 7 bits long. Using a character format of data without compression, the table itself occupies 17 bytes $\times 7$ tuples $= 952$ bits. Now it occupies only 49 bits.

We can use the conceptual distinction – abstract syntax versus concrete syntax in formal languages – and apply it to the set of relationships in a table. A table in compressed form represents the abstract syntax of the relationships in the original database. An uncompressed table holds the same relationships in concrete form. The compressed form holds the essence of the relationships, abstracting from the particular entities between which these relationships hold.

To translate these abstract relationships into concrete ones, it is necessary to go through a dictionary. The combination of dictionaries and the compressed table is shown in figure 2. An uncompressed database goes through a compression engine which splits it into two parts, the compressed tables and a set of domain specific dictionaries.

In their simplest form, these dictionaries can be simple lists of the values that occur in the domain. One would hope that the dictionaries would be smaller than the corresponding entries in the original table, since:

Figure 2: Translating a Database into the Compressed Form



- The original table would have contained fixed width fields. For a name, for example, this would have to be long enough to hold the longest name expected. Since most names would be shorter, a more compact representation could be used in the dictionary.
- In all columns but the simple primary key, there is likely to be repetition of field values. In our example, the strings 'CMOS' and 'TTL' occur repeatedly. Only one instance of these has to be stored in the dictionary.

In our previous example the compression might be as shown in table 2.

Table 2: Compression of CHIPSPEC

Raw				**Compressed**			
ep900	40	CMOS	Yes	000	00	0	0
ep310	20	CMOS	Yes	001	01	0	0
DMPAL16L8	20	TTL	Yes	010	01	1	0
DMPAL10L8	20	TTL	Yes	011	01	1	0
MPM5032	28	CMOS	No	100	10	0	1
CY2901	40	CMOS	No	101	00	0	1
CY2909	28	CMOS	No	110	10	0	1

We assume that the compressed code for a field is given by its position in the corresponding dictionary. The dictionaries are shown in Table 3.

After the compression, the fields of the relation can be concatinated into columns so no space is wasted by having partially used bytes.

Table 3: Dictionaries for CHIPSPEC

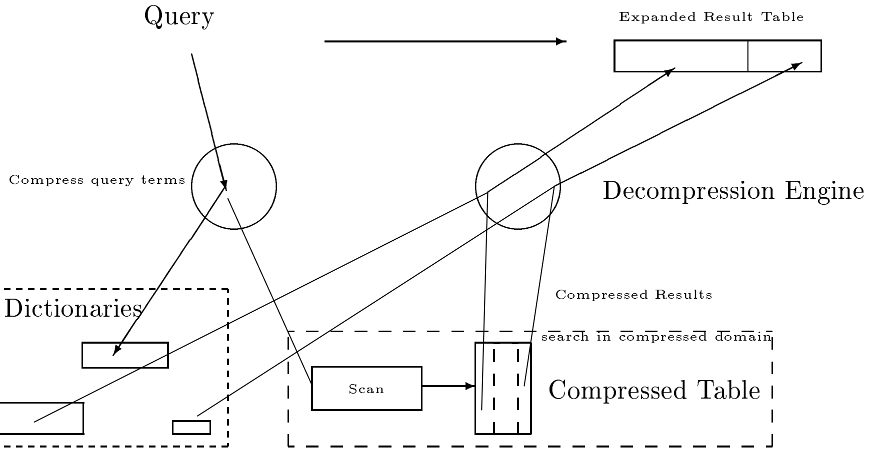
PART
ep900
ep310
DMPAL16L8
DMPAL10H8
MPM5032
CY2901
CY2909

PINS
40
20
28

Technology
CMOS
TTL

Programmable
Yes
No

Figure 3: Querying a Database in the Compressed Form



2.2 Query compression

When a query is performed, the query itself can be translated into the compressed domain.

Having compiled the query into the compressed domain, the amount of data that has to be moved into the CPU and compared for each selection operation is significantly reduced.

Clearly the final answer has to be converted from the abstract to the concrete representation. This involves a decompression phase in which the field codes are mapped to their associated keywords. But the computational cost of this will be borne only on the tuples that are returned as a result. Since this phase is anyway dominated by output delays in the screen handling software, the perceived cost of this decompression is likely to be minimal. The information transfers required to handle queries on a compressed database is shown in figure 3.

2.3 Second order compression

Suppose that we have a second relation that specifies the prices of chips in stock as a function of the number purchased:

PART	*Quantity*	*Price*
ep900	10	£ 18.42
DMPAL16L8	10	£ 7.04
DMPAL10H8	10	£ 6.92
ep900	1	£ 20.03
DMPAL16L8	1	£ 7.40
DMPAL10H8	1	£ 7.12
DMPAL16L8	100	£ 5.17
DMPAL10H8	100	£ 5.05

The domain from which the parts are drawn here is a sub-domain of those described in the CHIPSPEC relation. Since there are 3 parts listed as being in stock, first order compression column would require the use of 2 bit integers. This however would entail the establishment of a new dictionary to handle the partnames for the Prices relation. In the case of a primary key, the size of such a dictionary is almost certain to be greater than the total number of bits required to encode the column.

If, therefore, our objective is to minimize the size of the total database, it makes sense to determine which columns in the database can share dictionaries. In general these will be the columns that denote the same real world entities. This fact could either be captured by a set of type rules or, determined by statistical analysis of the database. We have developed a tool called SNET, which can automatically analyse an SQL database and determine which columns are potentially joinable and thus able to share dictionaries. We envisage that similar software could be incorporated into a database compression program.

Another potential area for compression is in the dictionaries themselves. We have oversimplified the dictionary representation by showing them in table 3 as just two dimensional arrays of characters. A dictionary must perform two mappings:

1. It must map fields to their encoded representation during the compression operation.

2. It must perform the reverse mapping from codes to literal values when parts of the relation are printed out.

It should perform these operations fast, with minimal waste of space. Further work is being carried out into combining these two mappings together in a way where the literal values held in the dictionary can be of an arbitary length. This would permit a compression scheme to operate efficiently on the literals.

Conclusion

The work we have described here arose from the conviction that the combination of datacompression with main store residence would give rise to a useful database architecture. This exists within a region of the cost/performance graph inaccessible to conventional disk-based databases and mainstore-based databases which do not use compression - faster than disk-based and less costly than main-store-only designs.

The problem in achieving this have been two-fold: first to achieve a great enough degree of compression, second to achieve this while retaining most of the performance characteristics of main-memory-based systems.

In particular, the difficulty is to retain good performance over all the range of necessary database functions, not simply one e.g. retrieval, and to do so without the system incurring reorganisation delays of arbitrary length at arbitrary times.

The work we have reported here shows that this is substantially possible, by appropriate trade-offs of performance versus degree of dynamic reorganisation.

The basic decision to store relations by column, each column being dynamically adjustable in both length and width, was taken on the grounds that this required less datamovement for each required adjustment than the corresponding row-based architecture. Reorganisation costs were further reduced by the multi-extent vector structure adopted which permits the structure to be incrementally adjusted by adding or removing a block, or widening blocks only as required. These decisions mean that the memory management system has to deal only with moderately-sized blocks, many, or most of which are of the uniform maximum blocksize, which can be readily handled by most operating systems.

The costs of making and adjusting indexes can be a severe overhead particularly during bulk loading operations. These were reduced by adopting generous initial allocation followed by a 'shrink to fit' at the end of the bulk operation.

Finally, as many operations as possible are conducted using the compressed representation of the data, this makes processing, and indeed I/O operations faster than on the corresponding uncompressed data.

Though we have been particularly concerned here with the architectural 'gap' between conventional and main-memory databases, the compressed representation described here has much wider application, and is in fact benefitial throughout the storage hierarchy, from processor cache, to archive medium. I/O bandwidths are effectively increased by the compression factor. We have carried out a number of paper design studies for organisations with extremely large databases in which compression would enable transfer from archive medium to be be reduced with consequent improvement in processing times.

In addition to the relation tables, the compressed vector structure is employed in indexes, and in constructing domain dictionaries. A particularly compact form of composite indexing has been adopted, based on Lloyd hashing. However, the compression reported here does not compress the representations of strings in the domain dictionaries. Despite this we have been able to demonstrate the significant compression factors given above. Dictionary compression can be added to further improve compression. To avoid a decompression overhead in frequent operations this is best applied at the end of bulk-loading, or as a database reorganisation operation.

The architecture described is certainly not a completely optimal one. Rather it is a minimal demonstration of the validity of the 'compressed and memory resident' database paradigm.

References

[1] D. Abramson. Hardware management of a large virtual memory. In *Proceedings of the 4th Australian Computer Science Conference*, pages 1–13, Brisbane, 1981.

[2] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[3] Goldstein,R. and Strnad,A. "The MacAIMS data management system" Proceedings of the ACM SCIFIDET Workshop on Data Description and Access, 1970.

[4] Lloyd,J. "Optimal partial-match retrieval" BIT, 20, 1980, 406-413.

[5] Pucheral,P. Thevenin,J. and Valduriez,P. "Efficient main memory data management using DBGraph storage model" Proceedings of the 16th VLDB Conference, Brisbane 1990, pp683-695.

[6] Bitton e. al, Benchmarking Database Systems - a systematic approach, in proceedings VLDB 1983

[7] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM SYSTEM/38 support for capability-based addressing. In *The Eighth Symposium on Computer Architecture*, pages 341–48, May 1981. Published in SIGARCH Newsletter, Vol 9, No 3.

[8] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level storage system. *IRE Transactions*, 2(EC-11):223–235, April 1962.

[9] Ashok Malhotra and Steven J. Munroe. Support for persistent objects: Two architectures. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Press, 1992.

[10] John Rosenberg and David Abramson. MONADS-PC - a capability-based workstation to support software engineering. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 222–231, 1985.

[11] John Rosenberg, James Leslie Keedy, and David Abramson. Addressing mechanisms for large virtual memories. Technical report, St Andrews University, 1990. CS/90/2. To appear in the Computer Journal, August 1992.

[12] Gordon Russell and [William] Paul Cockshott. Architectures for persistence. *Microprocessors and Microsystems*, 17(3):117–130, April 1993.

[13] G. Soltis. Design of a small business data processing system. *IEEE Computer*, pages 77–93, September 1977.

[14] E.J. Shekita T.J. Lehman and A.N. Wilschut. An evaluation of starbursts memory resident component. *Transactions on Knowledge and Data Engineering*, 4(6):555–567, December 1992.

[15] S. J. P. Todd. The peterlee relational test vehicle - a system overview. *IBM Systems Journal*, 15(4), 1976.

[16] Wang,C. and Lavington,S. "The lexical token converter - a hogh performance associative dictionary for large knowledge bases" Department of Computer Science, University of Essex, Internal Report CSM-133.

[17] Terry A. Welch. A technique for high performance data compression. *Computer*, 17(6), June 1984.