Computer Science • Vol. 10 • 2009

Krzysztof Dorosz*

USAGE OF DEDICATED DATA STRUCTURES FOR URL DATABASES IN A LARGE-SCALE CRAWLING

The article discuss usage of Berkeley DB data structures such as hash tables and b-trees for implementation of a high performance URL database. The article presents a formal model for a data structures oriented URL database, which can be used as an alternative for a relational oriented URL database.

Keywords: *crawling, crawler, large-scale, Berkeley DB, URL database, URL repository, data structures*

ZASTOSOWANIE DEDYKOWANYCH STRUKTUR DANYCH W BAZACH ADRESÓW URL CRAWLINGU DUŻEJ SKALI

W artykule omówiono zastosowanie struktur danych z pakietu Berkeley DB, takich jak: tablice z haszowaniem i b-drzewa do implementacji wysoko wydajnych baz danych adresów URL. Przedstawiono model formalny bazy danych zorientowanej na struktury pamięci, która może być alternatywą dla relacyjnie zorientowanej bazy danych linków URL.

Sowa kluczowe: *przeglądanie sieci, robot internetowy, Berkeley DB, baza danych URL, repozytorium URL, struktury danych*

1. Introduction

Within the beginning of the Internet there was always a need for an automatic browsing its resources for many purposes like: indexing, cataloguing, validating, monitoring, etc. Because of a todays large volume of the *World Wide Web* the term *Internet* is often wrongly identified with the single HTTP protocol service. The process of browsing the World Wide Web in automated manner is called *crawling* very likely by analogy between traversing the WWW using URL anchors[1] in the HTML pages to a bug crawling.

A crawling process is realised by dedicated software named *crawlers* or *robots*. Because of a large scope of possible applications, crawlers can vary with architecture,

* Institute of Computer Sciences, AGH University of Science and Technology, Krakow, Poland, dorosz@agh.edu.pl

[1] E.g. `<a href="..." ...>` tags.

complexity and needed resources. This article refers to the most challenging type of crawlers working with a large-scale enormously high volume content.

Every crawling system needs to have a control over traversing URL links what basically means it needs to know which URLs are already visited and which should be crawled next. This brings a need of using some databases to store and query URL links. As it will be shown further in this introduction efficiency of this database is a bottleneck of a high volume crawling system and there is a need of sophisticated solutions.

The most simple approach to this problem is to use a ready off-the-shelf relational database engine. In a practice it does not bring good efficiency while using with large-scale crawling systems. This article shows an example how to use Berkeley DB data structures for creating a relatively good replacement for a relational URL database which brings better efficiency and lowers usage of system resources.

1.1. Selected key issues of crawling

There are several issues of crawling that are worth to be mentioned before discussing an URL database architecture. Some of them points a relationship between a links repository design and a crawling system performance, others provides a conditions that had to be taken into consideration for an architecture of a database (features and/or restrictions).

This article does not attend to address any of presented in the following section detailed issues, focusing only at presenting crawling URL database architecture requirements.

1.1.1. Usage of system resources

Following the [7] every crawling system can be described using:

- number N of pages crawler can handle to the moment it cannot proceed further, because of resource restrictions,
- speed S of discovering web as a function of the number of pages already crawled,
- resources Σ (CPU and RAM) required to sustain downloading N pages at an average speed S.

Citing [7] „in most crawlers, larger N implies higher complexity of checking URL uniqueness, verifying robots.txt, and scanning the DNS cache, which ultimately results in lower S and higher Σ. At the same time, higher speed S requires smaller data structures, which often can be satisfied only by either lowering N or increasing Σ".

As can be seen in the citation mentioned above, designing a system which is able to cover large N with the designated S requires that the URL database must handle a volume of updates and inserts with given Σ resources. The database in this case is basically the main consumer of resources.

1.1.2. Avoiding spider traps

Spider traps (called also black holes) are specific endless loops of URL links. Some of them are natural consequence of using dynamic script languages (e.g. PHP) that supports providing parameters as a part of URL (the GET method). If used incorrectly every page can render a link to a next page with an another unique set of parameters (e.g. a calendarium of content, pagination, etc...) even if the further page does not bring any new information. Secondly spider traps can be set up intentionally be persons which want to spam web crawlers (high-density link farms, hiding content on a web page, and so on...). In general nature of a web pages graph connected through URL links should always be considered as an infinite node graph even refered only to a single host or domain. This assumption brings an elementary restriction for a URL scheduling algorithm that should be able to determine in the realtime which site contains interesting information for prioritising and should allow to crawl only the specified URL volume per host.

According to [7] previous implementations of crawling systems such as described in [3], [4], [5] and [6] were based on the BFS[2] strategy and does not take into consideration spider traps. It causes a fast convergence of a crawling process (about 3 to 4 BFS levels) to a limited number of hosts (domains) and an exhaust of resources what makes crawling useless the whole time.

To avoid such situation one can use different techniques suitable to a given purposes. The BFS tree height per host (or domain) can be precisely restricted by limiting a nodes per host number (or a domain per host number). This approach unfortunately bring a less accuracy when browsing the wide Internet because it is impossible to determine a single limit for the whole range of Internet hosts. Taking any number would cause that either this value will be not significant a for crawling process (if too big) or would cause that many information will be lost due to stopping a process too early on large web portals (if too little). Citing after [7] „Yahoo currently reports indexing 1.2 billion objects just within its own domain and blogspot claims over 50 million users, each with a unique hostname". Even if this number are not accurate by now, it can give a scale of a problem. Other technique involves using additional metrics like the PageRank (described in [1] and [2]).

For this reason URL databases should at least implement a method of storing information about a link depth in scope of a domain or host.

1.1.3. Information longevity

Every information is strictly connected with the media that contains it (e.g. text, picture) which is further connected with a web page and a given URL link. If the content of a web page changes after a while, it means that new information was added or/and old information was deleted. A crawling system that is aimed on keeping good relevance to current information over the Internet must deal with changes of a given

[2] BFS – Breadth-first search.

URL content. This problem is nowadays extremely difficult, due to highly dynamic nature of the Web. Olston et al. in [8] describe that a key role for dealing with an information longevity problem is to distinguish between an ephemeral and persistent content. They also define two important factors:

- *change frequency* – how often the content of a page is updated by its owner (also in [9], [10] and [11])
- *relevance* – how much influence the new/old page content has on a search result (also in [12] and [13])

Investigating a structure of current leading news websites one can see they are built from many media elements rather from a one (texts blocks, pictures, flash elements, etc...). This elements can rapidly change every refresh of the site, but at a given time they sustain randomized for a selected set of „news". This brings a noticeable difference because changing a little number of elements on the site should not be finally recognized as a content modification. Olston et al. [8] definitely understand the nature of this problem, because theirs research had been focused on *individual content fragments* rather then whole sites. Authors make an assumption that ephemeral information is not important „because it generally contributes a little to understanding the main topic of a page" what is discussable and definitely depends on crawling goals.

Problems described here proves that an URL repository should implement a method for re-enabling links after the given period of time for the main processing queue. This mechanism could be used by any refresh-policy algorithm which is able to determine a revisit time period for a given page.

1.2. Berkeleys DB data structures overview

Berkley DB[3] is an extremely performance open source[4] embeddable database eliminating a SQL overhead and interprocess communication by storing data in an application-native format. There are many wrapper libraries supporting Berkley DB for plenty of popular languages. Berkley DB offers basically four different simple data structures that will be described in this section. The description is mainly based on the Berkley DB Reference Guide[5]. A database supports operation of caching/storing each data structure to a separate files on a filesystem.

The common format for a Berkley DB structure element is a pair of $(key, value)$, which can be stored in every type of a structure. In many implementations of wrapper libraries this pair is one of formats:

- $(integer, string)$ – for the Recno structure,
- $(string, string)$ – for others structures.

[3] `http://www.oracle.com/database/berkeley-db/index.html`.
[4] Berkeley DB is currently owned by Oracle Company, but still remains open source.
[5] `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html`.

1.2.1. B-tree

This structure is an implementation of a sorted and balanced tree. A time complexity of searches, insertions and deletions is $\Theta(log_b N)$ where b is the average number of keys per page, and N is the total number of keys stored.

A very important fact is that the B-tree offers keeping *key*s order, what is significant for selecting the first/last element or browsing *value*s sorted by *key*s.

1.2.2. Hash

This is an implementation of the hash table based on the Extended Linear Hashing described in [14]. Basically a linear hashing provide a way for making a hash tables address space extensible (allowing it to grow or shrink dynamically) supporting any number of insertions or deletions. With reasonable assumptions time complexity of a search is $\Theta(1)$. No *key*s order is accessible in this structure.

1.2.3. Queue

The Queue structure implement a typical FIFO queue where records are allocated sequentially and directly mapped to an offset in a file. The Queue is designed for fast inserts and provide a method for deleting and returning a record from the head. The Queue structure supports only fixed-length *value*s.

1.2.4. Fixed and Variable-length records

The Recno is able to store records both fixed and variable-length with special logical record numbers as a *key*s. The Recno supports a number of additional features beyond those supported by the Queue, but the main advantage is supporting variable-length records.

2. Formal architecture for URL database

Summing up previously described restrictions and conditions an URL database should provide methods for:

- storing unique URL links, what requires a fast method for testing uniqueness of an URL in database,
- storing an URL depth for restricting a volume of links,
- selecting links for fetching from a pool using a prioritised order,
- confirming back to a database that a selected URL was finally crawled,
- guaranteeing, that if a URL was not confirmed in a given timeout, it will return to a pool immediately – that will avoid an URL droping without fetching, when a crawler unit selects a URL from a the pool and break down while processing it.

Described in this section activities are shown on Fig. 1. Solid lines refers to adding the new URL link to a database and are related with Section 2.2, Section 2.3 and Section 2.4. Das hed lines refers to selecting URL links from a pool and are related

to Section 2.4 and Section 2.5. Dotted lines refers to checking scheduled URL links for being correctly fetched or re-visited and are related to Section 2.4

2.1. Formalism

To reduce the complexity of a formal presentation of the described problem the following formalism will be used:

- $A[B]$ – is a data structure A that is addressed by a string key B,
- B *in* A – is a boolean value if the B is a defined key in the A data structure,
- (C, D) – is an ordered pair of elements C and D,
- $A[B] = (C, D)$ – is assigning a pair (C, D) to $A[B]$,
- $A[B] : C$ – is an extraction of the C value from a pair pointed by $A[B]$,
- $C \oplus D$ – is a string concatenation of the C and D.

2.2. Uniqueness test of URL

There is a given hash table U with non a empty key K represented by an URL address string. Let the $U[K] = (G, V)$. The uniqueness test of the URL K in a database is equal to checking the condition K *in* U.

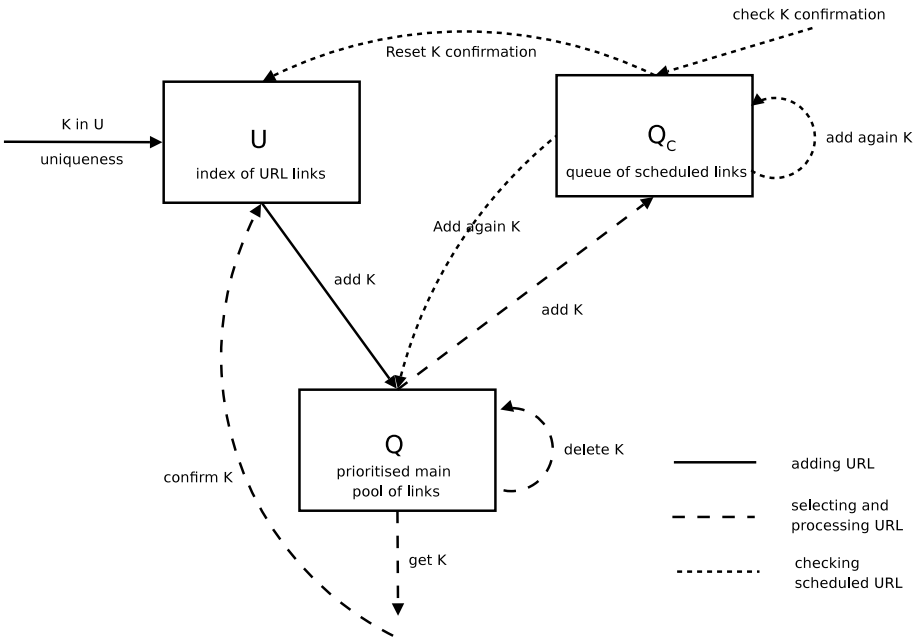

Fig. 1. Relations of data structures and functions

2.3. Depth of browse

Let the value G in the table U be a depth of the URL K. An expression $dn(K)$ represents a domain name string extracted from the K. There is a given pair of keys K and K', where the K' is a reference URL where the K was found. A value of the $U[K] : G$ can be defined as:

$$U[K] : G = \begin{cases} U[K'] : G + 1 & \text{if } dn(K) = dn(K') \\ 1 & \text{if } dn(K) \neq dn(K') \end{cases}$$

For starting links we assume the K' is an empty string $\varnothing$, what determine the $dn(K) \neq dn(K') \Rightarrow U[K] : G = 1$. Let the G_f be the final depth restriction for adding a new URL to a database. The K can be added to the $U \iff G \leq G_f$.

2.4. Prioritised queue with host load distributing

There is a given B-tree $Q[B] = K$ with the index $B = U[K] : G \oplus p(K) \oplus h(K)$ where:

K – is a given URL string,

G – is a depth of a given URL K,

$p(K)$ – is a one argument function that returns a priority of fetching a page K (a value of a priority is a number inversely proportional to the importance) converted to a fixed-width string padded by „0" (e.g. number 12 in four characters representation is „0012"; „0000" represents the highest priority, „9999" represents the lowest priority). The number of characters used in a priority representation depends on a resolution needed in the particular case,

$h(K)$ – is a fixed-length hash of the string K. Can be computed with any standard hash function such as the md5, sha1, etc.

Because the B-tree keeps a lexical order of string *key*s, selecting at any time the first element from the $Q[B] \Rightarrow K$ returns an element sorted in the following order:

1. links that are at the lowest depth level,
2. links that have the highest priority,
3. links hash order – basically it means a random distribution through hosts namespace.

This acts like a queue-like structure, where one can explicitly affect ordering, in this particular case it is the BFS with a prioritising among the BFS levels. Getting an element from this queue is managed by selecting the first $(key, value)$ pair from the Q and deleting the given B index from the Q.

Other links orders can be easily customised by concatenating the B index from different elements. For example for the $B = P \oplus U[K] : G \oplus h(K)$ selection order is the BFS for a constant priority with elements of the DFS[6] on priority changes.

[6] DFS – Depth-first search.

Sorting by hash values at the end gives a basic solution for distributing host load, but works only if the Q is not convergent to a very limited number of hosts. This method also does not guarantee a correctness of a distribution due to a characteristics of a hash method itself, so one have to know a limitation of this method.

2.5. Crawl confirmation

Let the V from the $U[K] = (G, V)$ be the boolean value representing a fact of fetching the page K. Every link K added to the U have a default value $U[K] : V = False$. After fetching the page K this value is changed to the $U[K] : V = True$.

2.6. Guaranteed URL fetching

Let the T_{cnf} be a given timeout before which a confirmation $U[K] : V$ of fetching the page K must be $True$. If the $U[K] : V = False$ the link K has to be available again in a links pool. There is a given confirmations B-tree $Q_c[B_c] = K$ with the index $B_c = T \oplus h(K)$, where:

K – is a given URL string,

T – is a timestamp in the Unix format (number of seconds, padded with zero for fixed-length representation),

$h(K)$ – is a fixed-length hash of a string K. Can be compute with any standard hash function such as the md5, sha1, etc.

Selecting the element K from a pool is evaluated with following steps:

1. Select the first element from the $Q[B] = K$.
2. Delete the element $Q[B]$.
3. Let the $B' = T_{now} + T_{cnf} \oplus h(K)$, where the T_{now} is a current Unix timestamp.
4. Add the $Q_c[B'] = K$.

Because the Q_c keeps a lexical order of *key*s, elements in the Q_c are ordered by a growing time of the T. The uniqueness of a key is guaranteed by a hash of the K, so links with this same time T are grouped together with a random order. The T represents a time in the future after which the page K should be checked if it was confirmed as a fetched page. Not often than a once in a pre-defined timeframe and before each link selection, the following procedure should be evaluated:

1. Get the first element from the $Q_c[B_c] = K$.
2. Split the B_c into a pair of values T and $h(K)$.
3. If the $T > T_{now}$ then stop processing.
4. Delete the element $Q_c[B_c]$.
5. If the $U[K] : V = True$ then jump to Step 1.
6. Add the element $Q[U[K] : G \oplus p(K) \oplus h(K)] = K$.
7. Jump to Step 1.

The Q_c can be understand as a logical queue of scheduled checks of page fetch confirmations. In Step 1 the described algorithm waits for elements that are ready to

check. If the first element points a time moment in the future – Step 3 – (it is not ready to be checked) procedure can stop because a *key*s order in the Q_c guarantee, that next elements points at least this same time moment or a further moment. If an element is ready it can be deleted from the Q_c – Step 4 – and checked if fetching a confirmation for the given page K was $True$ – Step 5. If not, the procedure put back the page K into the main pool (Q) – step 6.

2.7. Time delayed re-crawling

Using the B-tree instead of the Queue for implementation of the $Q_c[B_c] = K$ brings a possibility of inserting elements with a random T values from the B_c index. Such elements will be sorted in a lexical order of the key B, what can be easily used in an implementation of a re-crawling scheduler.

Let the $r(K)$ be the function of a one argument K returning a time period after the page K should be re-crawled. A modification (selected with the **bolded** font) of the checking algorithm described in the previous section brings a simple and elegant solution of a time delayed re-crawling:

1. Get the first element from the $Q_c[B_c] = K$.
2. Split the B_c into a pair of values T and $h(K)$.
3. If the $T > T_{now}$ then stop processing.
4. Delete the element $Q_c[B_c]$.
5. If the $U[K] : V = True$ then **jump to Step 8**.
6. Add the element $Q[U[K] : G \oplus p(K) \oplus h(K)] = K$.
7. Jump to Step 1.
8. **If not** $r(K) > 0$ **then jump to Step 1.**
9. **Let the** $U[K] : V = False$.
10. **Let the** $B' = T_{now} + r(K) \oplus h(K)$**, where the** T_{now} **is a current Unix timestamp.**
11. **Add the** $Q_c[B'] = K$**.**

After a successful check on a page fetch status – Step 5 – the link K can be added back to the Q_c. The $r(K)$ can be set to a non-positive value to prevent a re-crawling this page in future – Step 8. Setting the value $U[K] : V$ back to $False$ – Step 9 – and adding the K back to the Q_c – Step 11 – cause that after the $T_{now} + r(K)$ time moment the link will be added again to the main pool Q.

3. Benchmark

The benchmark was performed on an IBM machine with the Power5+ processor (8 cores – 1648.35 Mhz each) and a 12 GB RAM memory. The described model was implemented twice in the Python language (version 2.x). The first implementation was an URL database prototype using the DB Berkley data structures and was based

on the bsddb3 module[7]. The second implementation was a prototype of relational database URL links repository prototype and was based on the PostgreSQL (version 8.2) with the standard pg module. For time measurements the Python timeit module had been used. The representative test of both URL links repositories was performed by a measurement of a delay of inserting single new URL depending on the database volume. The database start condition was 1000 000 links already inserted. The next 3000 000 links have been added until the benchmark. URL links were generated randomly. The set of domains contains 10 000 different random domains. Random links had been generated by concatenating `http://` prefix with a random domain from the set and a random length path from a characters set: `[a-zA-Z/-_]`. Characters in the characters set were weighted to achive a better similarity to real links (e.g. `http://dsurfrd.erge.er.pl/FVSdfER-fb/d_dfb/`). The randomization time had no effect on the delays measurement.

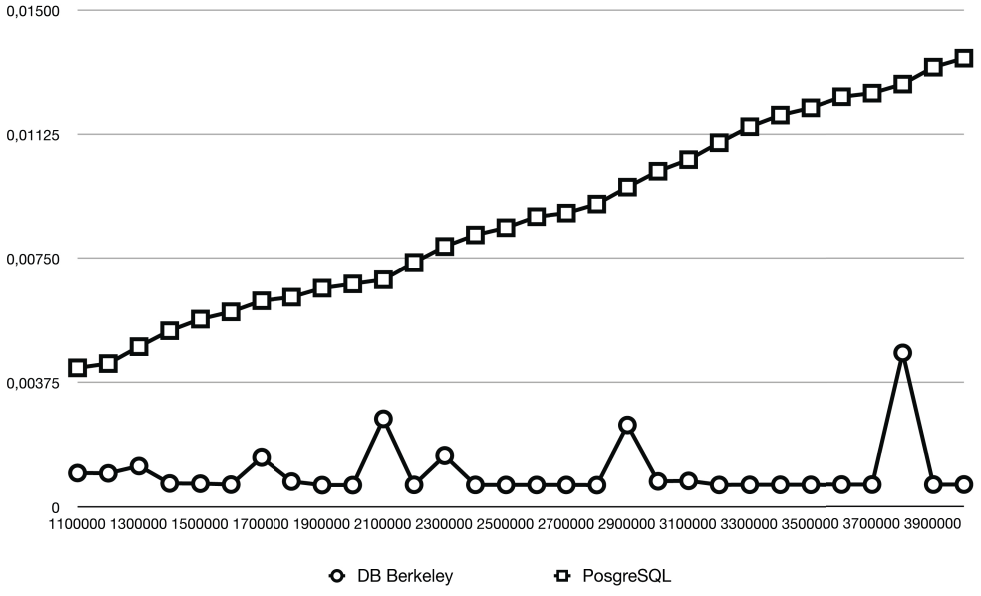

Fig. 2. An avarage add URL delay [s] while adding 3000 000 links dependent on the database volume

Average delays were measured for every 100 000 set of URLs inserted to database. Delays of inserting one link are presented on Fig. 2. X-axis labels are describing the total volume of the database. Y-axis labels are delays in seconds of adding one link at the given volume. As can be seen from presented results a DB Berkeley implementation gives better performence then the PostreSQL for use in storing URLs.

[7] `http://pybsddb.sourceforge.net/`.

4. Conclusion

The presented method for implementing an URL database based on data structures allows to design a system that meets all given requirements described in Section 1.1. As can be seen defining a formal design of a database strictly defines a specific crawling policy. Any customisation to a crawling policy should be provided as a variation to the formal design of a database, but it will keep the high performance of a data structure based implementation.

References

[1] S. Brin, L. Page: *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* in Proc. WWW, pp. 107–117, 1998

[2] S. Brin, L. Page, R. Motwami, T. Winograd: *The PageRank citation ranking: bringing order to the web.* Proceedings of ASIS'98, 1998

[3] A. Heydon, M. Najork: *Mercator: A Scalable, Extensible Web Crawler.* World Wide Web, vol. 2, no. 4, pp. 219–229, 1999

[4] M. Najork, A. Heydon: *High-Performance Web Crawling.* World Wide Web, vol. 2, no. 4, pp. 219–229, 2001

[5] V. Shkapenyuk, T. Suel: *Design and Implementation of a High-Performance Distributed Web Crawler.* in Proc. IEEE ICDE, pp. 357–368, 2002

[6] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. R. G. Wesley: *Stanford WebBase Components and Applications.* ACM Transactions on Internet Technology, vol. 6, no. 2, pp. 153–186, 2006

[7] H.-T. Lee, D. Leonard, X. Wang, D. Loguinov: *IRLbot: Scaling to 6 Billion Pages and Beyond.* Texas A&M University, Tech. Rep. 2008-2-2, 2008

[8] C. Olston, S. Pandey: *Recrawl scheduling based on information longevity.* conf/www/2008, pp. 437–446, 2008

[9] J. Cho, H. Garcia-Molina: *Effective Page Refresh Policies for Web Crawlers.* ACM Transactions on Database Systems, 28 (4), 2003

[10] E. Coffman, Z. Liu, R. R. Weber: *Optimal robot scheduling for web search engines.* Journal of Scheduling, 1, 1998

[11] J. Edwards, K. S. McCurley, J. A. Tomlin: *An Adaptive Model for Optimizing Performance of an Incremental Web Crawler.* In Proc. WWW, 2001

[12] S. Pandey, C. Olston: *User-centric web crawling.* In Proc. WWW, 2005

[13] J. Wolf, M. Squillante, P. S. Yu, J.Sethuraman, L. Ozsen: *Optimal Crawling Strategies for Web Search Engines.* In Proc. WWW, 2002

[14] W. Litwin: *Linear Hashing: A New Tool for File and Table Addressing.* Proceedings of the 6th International Conference on Very Large Databases (VLDB), 1980