

# Optimizing firewall performance

Anssi Kolehmainen  
Helsinki University of Technology  
anssi.kolehmainen@hut.fi

## Abstract

Firewalls are one key factor in network performance. If they can't process their rules fast enough then the whole network slows down. Firewalls are also required part in every network so special attention must be paid to their packet matching algorithms which we study in this paper along with other rule optimization methods. We find out that there is no single best algorithm for every case. Therefore one needs to know all the alternatives to choose the right algorithm for the application at hand.

KEYWORDS: Firewall, packet filtering, rule optimization

## 1 Introduction

Our computers are constantly under attack from hackers on the Internet. The only way to stay safe is to have a good firewall, up-to-date software and some common sense. Firewalls have become necessary devices in every network and nearly in every computer. Increased network speeds create huge processing requirements for traffic filtering because firewalls must make decision for every packet whether to allow or deny them. Typical home firewalls might have only few dozen rules and a couple of IP addresses on the home side whereas corporate firewalls might have thousands of rules and hundreds of IP addresses to take care of.

Every time a packet arrives, a firewall must examine its rules and decide whether to allow or drop it. A simple approach is to scan every rule in priority order until a match is found. However that is very inefficient. For example, if we take a normal 100mbps connection we get about 8 000 packets per second (assuming maximum Ethernet frame size) which means a firewall has about 100 microseconds to make the filtering decision to keep up with full bandwidth. Current computers have memory access latencies around 50-100ns which leaves opportunity for 1000-2000 memory accesses per packet in optimal case. Of course there are some other processor level optimization (i.e. caches and prefetching) which can provide some extra performance but then again their real world effect is somewhat unpredictable. Those thousands of memory accesses per packet might seem plenty but when you count in all the other factors (operating system for instance) that eat away the performance and the real CPU processing requirements that doesn't leave out too much headroom. Even still that might be enough for small home firewall with dozen of rules but then again corporate firewalls might have thousands of rules and even higher traffic. Therefore it is imperative that we optimize firewalls as

much as possible.

Packet classification algorithms have been studied for several years but not as long as some other central techniques in computing. Gupta's article [8] covers many of the previous algorithms and has somewhat similar target like this paper. However, many new ideas have been presented since 2001 and there hasn't been a comparative study of them. This paper collects a group of old and new algorithms, describes them shortly and tries to compare them against each other.

In the next chapter we discuss the background of packet filtering and take a look at previous research and in chapter 3 we look at some newer methods. Finally there is analysis in chapter 4 and conclusions in chapter 6.

## 2 Background

Firewalls work by inspecting different fields in headers of the packet, finding the matching rule and doing the action specified in the rule. Common firewalls (and rules) examine at least IP addresses, next protocol type (TCP or UDP) and in case of TCP or UDP their respective port numbers. That usually gives five different variables (source & destination address, protocol, source & destination port) to work with. Different header fields are commonly referred as *dimensions* since each packet can be thought as a point in n-dimensional space with rules being hypercubes and packet matching as algorithm of finding out in which hypercube a point is in. Therefore many of the algorithms have graphical background instead of classical *linear search* background. However some graphical-based algorithms work only on two dimensions. Usually that can be chosen to be source and destination IP addresses which limit the remaining search base considerably.

Firewalls can also examine many other header fields in packets but that is not considered in this study. Many of the algorithms have more general versions of them which can examine n-dimensional values. It is up to the implementation of the firewall to choose what fields to process. Moreover, since some header fields are 1-bit flags (or similar with very small range) it is not practical to use these wide range matching algorithms for them. One possibility is to use these algorithms for matching source and destination IP-addresses (and maybe TCP/UDP ports) and then different algorithms for the remaining fields.

Some firewalls can modify packets based on the rules but this study concentrates only on simple allow/deny results. However, most of the algorithms try to find the actual rule which matches the packet and don't make any assumptions on the content of the rule. That leaves possibility any kind

of rules (i.e. packet modifying rules). Furthermore, most firewalls tend to be stateful which means that they remember which packets have previously passed through and can make decisions based on that (e.g. allow all packets to both directions for already established connections). This study concentrates only on stateless packet classification algorithms since they are required in every firewall and state tracking adds more complexity. One way of implementing state tracking is to add a new rule each time a new connection is made. This however leads to rapid increase in rule count which in turn lowers performance.

Key factor with firewalls is that they must always come to the same conclusion for same packets (i.e. behave deterministically). Possibilities for indeterministic behavior are, for example, conflicts and anomalies in the rules. Methods for detecting and removing such errors have been discussed in few articles [11, 1]. Removing any such clear *errors* should be done before any other optimization steps. Also since their removal might need human interaction (to decide which rule is the right one) this should be done when adding the rules so immediate feedback could be given. Other ways to solve these errors include prioritizing the rules so that we believe either the first or the last to be the correct action. Such assumptions can however lead to different behavior than expected.

As mentioned in the introduction, algorithms have been studied for several years. One published article by Gupta [8] collects many different algorithms and is very similar to this paper. However, Gupta's article was made in 2001 and after that some new ideas have been presented which I study in this paper. Also Gupta's article doesn't present any conclusions, it only describes algorithms and their time and storage complexities.

### 3 Optimization methods

Optimization can happen in many places. First possibility is when rules are added to the firewall. This is somewhat rare event (when compared to filtering packets) so it can use more resources. However it shouldn't interrupt normal operations for too long. Steps done in this first phase include anomaly detection [1], rule aggregation and rearrangement [2] as well as other initialization steps for algorithms (like precalculating values and building trees).

Second place for optimization is the rule checking. Every time packet arrives some algorithm must be used to check the rules. Algorithms should emphasize quick runtime so firewalls can keep up with the traffic. Downside is that firewalls may be external devices with very little memory so that puts some limits on the algorithms. Although when speaking of external devices there are completely hardware based firewalls which take advantage of specialized processors. For example Ternary CAMs [8] can achieve  $O(1)$  runtime with specialized hardware but they are impractical to implement in standard computers or firewalls since those chips cannot be used for any other processing. However, in very demanding situations custom ASICs (like Ternary CAM chips) are required.

Basic problem on choosing the right algorithms is the speed-memory trade-off. To get results faster you need more

memory for precalculated values. Normally this might not be such a big problem since computers have gigabytes of memory but since firewalls run in the background or they can even be external embedded hardware devices that don't contain much memory they can run out of memory for more advanced algorithms. Bigger specialized corporate firewalls are different story though. They might even contain specialized chips that work in totally different way.

One common thing between most of the algorithms presented here is that they operate on prefixes rather than on complete data. For example when considering an IP address it is rather expensive to use all 32-bits when you have to build trees that have every possible value. It is much simpler to use only first few bits (i.e. prefix) of the address for the main algorithm and do rest of the comparisons with linear search. It is also possible to convert ranges into prefixes. That however usually leads to multiple prefixes which all must be considered.

Following methods are separated into five different categories. First category concentrates on one-time methods and rest on different approaches for finding matches on incoming packets as quickly as possible.

#### 3.1 Preprocessing methods

Man-written firewall rules tend to be non-optimal. Mistakes can be categorized into four different classes: shadowing, redundancy, correlation and irrelevance [1]. *Redundancy* means a former rule matches all packets which some later rule would match and both rules have same action. Therefore the later rule is never executed and could be removed without any changes in the outcome. *Shadowing* is basically same as redundancy except that shadowed rules have different actions than their former rules and therefore cause irregularities in firewall action if their order is reversed. *Correlation* is similar to shadowing expect that the two rules don't overlap completely so neither can be directly removed. Their order of execution however changes the end result for some packets. Finally *irrelevance* means that firewall includes rules which will never match. For example rules might contain IP-addresses that never cross the firewall.

Having these anomalies gives limits to the filtering algorithms since they must maintain some order between rules. For example one simple optimization is to put all rules in decreasing order of probability. If this is done with firewall with shadowing and correlation anomalies it generates false results. Therefore such rules cannot be ordered in optimal fashion for linear search.

Al-Shaer and Hamed have studied [1] how often these errors are made. They find that 0% - 12% of man-written rules contains these errors. This might not seem much but considering that even experts make some mistakes leads us to believe that typical home users might make really many errors. If amount of rules can be decreased by 10% with simple preprocessing that would yield similar (or higher) performance gains. The other important factor is security. If home users make so many errors in general they might also make errors which compromise the firewall. Therefore firewall administration programs should be more intelligent in spotting those errors.

Hari et al. [11] proposes algorithms for eliminating such errors. One good side of their algorithm is that it has run time of  $O(w^2)$  (where  $w$  is the width of field) which means that it scales well with large number of rules. However they point out that with precomputation filter update time rises to range of  $O(n)$ . They have experimental results with database of 10 000 rules which show that the conflict detection takes about 6 microseconds (on 200MHz Pentium Pro workstation).

Another way for optimization is *rule rearrangement* [2]. It's usefulness depends somewhat on the used algorithm but in can give some improvements. Ideal case of rearrangement assumes that there are no conflicts between rules (e.g. shadowing or correlation) so it doesn't matter in which order the rules are processed. Then the rules can be arranged for example by the source IP-address and then simple binary search can be used to find the matching rule giving instant reduction from  $O(n)$  runtime to  $O(\log(n))$  runtime. Dynamic methods (in chapter 3.5) are similar but they sort rules in a different way.

*Directed acyclic graphs* [4] can be used for finding more optimal order for rules. The paper states that finding the optimal arrangement is NP-hard problem. Basic idea is to sort all rules in decreasing order of probability while retaining integrity (since some rules may shadow or correlate with each other). DAGs are used to represent the firewall so that correct order is kept. The paper gives a simple sorting algorithm which probably has around  $O(n^2)$  worst-case runtime. On a sample list it yields 11% fewer comparisons on average. It is also worthy to note that this method has been patented [5]. These days one can never be sure what things can be used completely freely.

*Firewall decision diagrams* (FDD) [7] are more of an formalization of the firewall than a direct optimization algorithm. If we can describe firewall using simple logical closures we have an many logic tools for reducing rules [14]. The paper [7] describes five different algorithms for turning user specified FDD into list of simple firewall rules. Those algorithms compress the rules to find smallest number of required rules.

### 3.2 Classical methods

The simplest method is *linear search*. We have a list of all rules sorted in priority order (first rule has biggest priority) and we examine each rule in order until a match is found. Very simple to implement but it has  $O(n)$  runtime. However  $O(n)$  storage requirement is optimal. With small enough datasets this might be the best algorithm.

*Tries* [8] are tree-like structures which have values on the edges rather than in the nodes. Therefore tries may also have many children in some nodes (if edge values are long like full IP-addresses). Commonly these tries have address prefixes as their values which are searched bit-by-bit (i.e. these tries look something like binary trees). There are also multiple tries included in packet matching. One trie (the F1-trie) is used for first dimension (e.g. source IP address prefix) and then other tries (F2-tries) are used for second dimension (e.g. destination IP address prefix) and so on. There are usually links from F1-trie to F2-tries in some points (which depend on the actual algorithm).

$N$	number of the rules
$d$	number of dimensions used in matching
$W$	(IP address) prefix length
$S$	hardware word size in bits
$\alpha, l$	algorithm specific tunable parameters

Table 1: Different variables used in  $O()$ -notation

*Hierarchical trie* [8, 13] (a.k.a. backtracking trie) has one F1-trie for first dimension and multiple F2-tries for second dimension and so on. When matching a packet we want to find the longest matching prefix. We do this by following the trie from root bit by bit. Some nodes might contain links to other nodes and in some cases we must backtrack to find the right path. Hierarchical tries have storage complexity of  $O(NdW)$  which is larger than with linear search but  $O(W^d)$  runtime is much better. *Set-Pruning trie* tries to eliminate backtracking and achieves  $O(dW)$  runtime with  $O(N^d dW)$  storage. However  $O(N^d)$  worst-case upgrade complexity is rather large.

*Grid-of-tries* [8] extends hierarchical tries by precomputing switch pointers in some nodes. That results in  $O(W^{d-1})$  runtime,  $O(NdW)$  storage and  $O(NW)$  updates. However grid-of-tries is limited to only two dimensions. Therefore it works well together with hierarchical trie.

*Bit vector linear search* [2] is also based on tries but it has  $N$ -bit vector in each node. The bit vector has a bit on for every rule that matches corresponding node. We have one such trie per every dimension. To get matching rules we have to calculate intersection of  $N$  bits between  $d$  vectors, thus we get  $O(\frac{Nd}{S})$  (where  $S$  is the word size in bits) plus cost of traversing the trie. Storage required is space for  $d$  tries with depth of  $W$  and  $N$ -bit vector in each node (if sparse trees are not used), thus we get  $O(\frac{N^2 d}{S})$ .

*Aggregated bit vector* [2] is improvement over bit vectors. Basic idea is to reduce size of bit vector since with large rule sets it wastes much space and is expensive to calculate. Aggregation is done by ORring  $A$  bits into one bit in the bit vector. The paper [2] suggests that  $A$  should be the word size. ABV algorithm requires  $O(N^2 d)$  preprocessing time. Storage required over standard bit vector is  $O(\frac{N^2 d}{AS})$ . Updates have same complexity as with bit vector.

### 3.3 Geometric methods

*Cross-producting* [8] is based on creating  $d$ -dimensional hypercube by composing different 1D range lookups. Different compositions of ranges in  $d$  dimensions are stored as tuples  $(r_d^i, \text{e.g. } (r_1^1, r_2^1), (r_1^2, r_2^2) \dots)$  with matching rule. Worst case storage is huge  $O(N^d)$  if ranges comb the whole area. Lookup takes only  $O(d \log(W^2)) = O(dW)$  time.

*2D classification scheme* [8] is composition of F1-trie (as in hierarchical tries) and range lookup lists (for second dimension). Storage complexity is only  $O(NW)$  with lookups taking  $O(W \log N)$ .

*Area-based quadtree* [8] is somewhat similar to cross-producting but instead of storing ranges it splits the 2D-square into four parts (corresponding to prefixes 00, 01, 10 and 11). Then search is done using a quaternary trie

(like binary tree but with four children). This algorithm has  $O(NW)$  storage complexity,  $O(\alpha W)$  lookup time and  $O(\alpha^\alpha \sqrt{N})$  updates where  $\alpha$  is tunable parameter representing how many divisions are done.

*Fat inverted segment tree* (FIS-tree) [8] is a modified segment tree for 2D lookups. It has  $O((l+1)W)$  lookup complexity and  $O(l \times N^{1+1/l})$  storage complexity. Parameter  $l$  can be tuned depending on wanted speed-memory trade-off. The article [8] mentions that 4-60K rules require 5MB of storage and less than 15 memory accesses per query.

### 3.4 Heuristic methods

*Recursive flow classification* (RFC) [8] works by preprocessing tables which shrink input data into fewer bits which are then again used to lookup other tables. This is done recursively and the final step yields the correct action. RFC is said to work on gigabit lines with thousands of rules. Lookup complexity of  $O(d)$  and storage requirement of  $O(N^d)$  support this.

*Hierarchical intelligent cuttings* (HiCuts) [8] is somewhat similar to cross-producing and area-based quadtree but instead of fixed divisions it tries to find optimal half cuts to minimize search tree. Paper says that with about 1700 rules HiCuts requires less than 1MB of storage and 20 memory access per query. Lookup and storage complexities are same as with RFC.

*Tuple spaces* [8] store matching rules with tuples representing prefix lengths. Advantage is gained through hashed tuple storage which is efficient to lookup. Also storage complexity is only  $O(N)$  since rules are stored only once. Lookups take around  $O(N)$  if amount of tuples stays small. Multiple hash collisions reduce performance considerably. However updates require only single entry and therefore work in  $O(1)$  time.

### 3.5 Dynamic methods

*Early traffic rejection* [10] works by periodically (i.e. every  $n$  packets) building a list of most frequently hit rejection rules and then first compares packet against that list before trying normal packet filters. The paper gives estimations of 7% to 40% gains on number of comparisons when using early rejection.

*Statistical filtering tree optimization* [10] works by modifying the search tree dynamically to include most frequently used field values in the shortest path. Idea behind this is that majority of traffic uses the same rules and therefore it is beneficial to check them first. The paper proposes keeping statistics of how often rules are hit and then building alphabetical trees for each search field based on that information. In real life case that resulted in 40% to 60% gains over binary search when looking up port numbers and 10% to 40% gains when looking up IP addresses. Ratios vary slightly when using different update frequencies. Storage complexity for lookup trees is  $O(N)$  and lookup time  $O(N \log N)$ .

*Dynamic rule ordering* [9] is also based on the fact that most of the Internet traffic is caused by a few long-lived flows. The paper proposes an  $O(n^2)$  algorithm for rule ordering with  $O(n)$  storage. However it is stated that in real-

life scenarios run-time is fraction of worst-case scenario. After initial ordering packet matching involves few extra calculations and periodic updates. Test traffic shows 40% to 60% matching reduction when using this technique.

*Data mining techniques* [6] have similar approach as other dynamic methods, find the most often used rules and try to reorder. However the method is somewhat different. Whereas the other methods work *online* this methods works *off-line* by examining firewall logs and rules. End results are similar but off-line methods require some human interaction so the results can be put into use.

## 4 Analysis

There are many different methods for optimizing firewall rules. Basically, they can be divided into three groups. Methods from first group are used only once when rules are changed. First group contains algorithms which try to optimize out the unnecessary rules and perhaps order them in more optimal order. The second group contains algorithms and methods for actual packet matching and the third group has algorithms for learning what kind of traffic is on the network and reordering the rules based on that (for second group algorithms which use ordered rules). Third group also contains some early rejection methods which reduce the traffic for second group methods.

Some preprocessing algorithms seem definitely useful for any kind of filtering algorithm. Reducing number of rules gives performance gains in every case. Rule rearrangement helps only if the algorithm results in multiple rule comparisons. Most of these algorithms tend to find smallest possible group of rules since their full comparisons are somewhat expensive.

On general level heuristic methods seem to have best performance and storage ratio if storage isn't clearly limited. However they can be hard to implement right and might not always provide best results. Simpler algorithms (like tries) give good enough results on smaller number of rules and if updates don't happen too often. Also some combination algorithms (like 2D classification) give good results. In most cases it should be enough to use 2D methods for example source and destination IP addresses and then do linear search on resulting rules.

Another important factor that hasn't been looked much before is speed of the updates. In traditional stateless firewalls updates are rarely needed (maybe once a day or even less) so that performance impact is negligible. However, in stateful firewalls (and home environments) rule updates are required more often. Worst case home scenario might require new rule for every new connection and with some peer-to-peer file sharing that might result in dozens (or even hundreds) of rule additions every second.

Corporate environments tend to be more stable but they have multiple computers. If stateful firewall is implemented with rule additions then it would also result in multiple updates every second (depending on the size of the network and whether services are offered to internet).

Table 2 gives summary of different packet matching algorithms discussed in this paper. They are approximately

Algorithm	Lookup	Storage	Updates
Linear search	$O(N)$	$O(N)$	$O(1)$
Tuple space search	$O(N)$	$O(N)$	$O(1)$
Hierarchical tries	$O(W^d)$	$O(NdW)$	$O(d^2W)$
Grid-of-tries	$O(W^{d-1})$	$O(NdW)$	$O(NW)$
Set-pruning tries	$O(dW)$	$O(N^d dW)$	$O(N^d)$
Cross-producing	$O(dW)$	$O(N^d)$	?
Linear bit vector	$O(Nd/S)$	$O(NdW^2/S)$	?
Statistical trees	$O(N \log N)$	$O(N)$	?
2D classification	$O(W \log N)$	$O(NW)$	?
Area based quadtree	$O(\alpha W)$	$O(NW)$	$O(\alpha^\alpha \sqrt{N})$
FIS-tree	$O((l+1)W)$	$O(l \times N^{1+1/l})$	?
RFC	$O(d)$	$O(N^d)$	?
HiCuts	$O(d)$	$O(N^d)$	"fast"
Bitmap-intersection	$O(dW + N/memwidth)$	$O(dN^2)$	-
Ternary CAM	$O(1)$	$O(N)$	-

Table 2: Summary of filtering algorithms

sorted in increasing performance order. The table largely resembles the one in Gupta's article [8] but that is inevitable since this paper contains so many algorithms from that same article.

It is also possible to gain performance with parallelization [3]. This however requires either use of some sort of load balancing for  $n$  similar firewalls or communication between different filtering pipelines. Then again RFC keeping up with 10Gb/s in hardware solution [8] seems *enough* for everyone. Although that has some resemblance to common quote "*640Kb should be enough for everybody.*"

There are also pure hardware solutions which have different possibilities. For example *Ternary CAMs* [8] have runtime of  $O(1)$  since they can compare multiple rules in parallel. However since such hardware is limited it needs its own optimizations [12].

## 5 Further research

Many methods have been found and tested for optimizing firewall rules but the *one method to rule them all* is still missing. Further research could give us completely new ways to firewall optimization. One interesting topic would be to focus on different areas. For example, graphical algorithms might yield good results.

As we have seen, specialized hardware like CAMs have excellent performance by comparing multiple rules at once. Quantum computers have same kind of characteristic behavior but being still rather impractical it's hard to find new algorithms for them. Moreover, packet matching requires mainly memory accesses so quantum computing might not help here. Of course there will be a completely new area of algorithm studies once we really start getting quantum computers.

A second main area of research would be actual performance testing of these algorithms. General algorithmic worst-case estimates might not mean anything in real life because situations can be different. One could build a testing framework where it would be easy to implement differ-

ent algorithms and then test their performance in simulated real-life environment. These environments should contain a number of different situations, some with few rules and some with huge number of rules. Also, performance with ruleset updates should be tested.

It is worthy noting that none of these algorithms make any statements on stateful firewalls. They all consider only simple static rules. In real life, two way communications are required and that basically requires maintaining state of connections (or creating specialized rules for every connection). State tracking adds more complexity to firewalls and efficient solutions might require the use of completely different algorithms. One solution for state tracking could include automatically adding rule for every connection and then removing them when connection is closed. Also home devices could request new firewall rules for new connections through Universal Plug and Play -interfaces.

## 6 Conclusion

To sum it up, some lookup algorithms emphasize minimizing memory usage and others minimizing execution time. Simply by looking at the theoretical bounds of the algorithms we don't get any clear winners so the best algorithm depends on the actual situation. Some algorithms could be optimized to suit more special needs (e.g. home firewall which has few addresses on the other side) and some algorithms are better suited for on-the-fly statistical optimization than others. One important factor might be update speed since some scenarios require multiple updates and some algorithms have really heavy updates. Finally there is the programming complexity issue. Why spend countless number of work-time trying to code the fastest possible algorithm when it doesn't give any real advantages over simpler algorithms. Therefore best algorithm should be decided on per-case basis.

## References

- [1] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE Infocom*, 2004.
- [2] F. Baboescu and G. Varghese. Scalable packet classification. In *Networking, IEEE/ACM Transactions on*, 2005.
- [3] R. J. Farley. Parallel firewall designs for high-speed networks. Master's thesis, Wake Forest University, December 2005.
- [4] E. W. Fulp. Optimization of network firewall policies using directed acyclical graphs. In *IEEE Internet Management Conference*, 2005.
- [5] E. W. Fulp. Methods, systems and computer program products for network firewall policy optimization. World Intellectual Property Organization, WO 2006/105093 A2, October 2006.
- [6] K. Golnabi, R. K. Min, L. Khan, and E. Al-Shaer. Analysis of firewall policy rules using data mining techniques. In *IEEE Network Operations and Management Symposium*, 2006.
- [7] M. G. Gouda and A. X. Liu. Firewall design: Consistency, completeness and compactness. In *IEEE International Conference on Distributed Computing Systems*, 2004.
- [8] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, pages 24–32, March/April 2001.
- [9] H. Hamed and E. Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *ACM Symposium on Information, computer and communications security*, 2006.
- [10] H. Hamed, A. El-Atawy, and E. Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. *unpublished*.
- [11] A. Hari, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *IEEE Infocom*, 2000.
- [12] T. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, and N. Dulay. Compiling policy descriptions into reconfigurable firewall processors. In *IEEE Symposium on Field-Programmable Custom Computing Machines, Proceedings of*, 2003.
- [13] L. Qiu, G. Varghese, and S. Suri. Fast firewall implementations for software and hardware-based routers. In *Network Protocols*, 2001.
- [14] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *Deductive and Object-Oriented Databases*, pages 204–221, 1993.