

THE LOST ALGORITHMS

Volume Three — Databases

By

MARTIN CARR

INDEPENDENT PUBLISHER

Copyright © 2026 by Martin Carr.

All rights reserved.

No part of this book may be reproduced, digitalized, or transmitted in any form without written permission from the publisher, except in the case of brief quotations embodied in critical reviews.

Publisher: Independent Publisher

ISBN Placeholder

TABLE OF CONTENTS

Preface: The Machinery Under the Question	4
How to Read This Book	7
A Note on Patents and the Public Domain	9
Part I — The Index: Finding Without Looking	11
The Balanced Tree (B-tree Indexes)	12
The Ordered Shelf (Sort-Order Indexes)	19
Many Keys at Once (Composite Indexes)	24
The Word Map (The Inverted Index)	29
The Table That Grows Itself (Extendible Hashing)	34
Part II — Bitmaps: Sets as Bits	39
Sets as Bits (Bitmap Indexes)	40
Squeezing the Bits (Compressed Bitmaps)	46
Part III — The Query: Answering Fast	52
Zipping Two Tables Together (The Sort-Merge Join)	53
The Answer Kept Ready (Aggregate Indexes)	58
Questions on a Moving River (Event Filtering and Windowed Aggregation)	63
Asking a Better Question (Query Refinement and Relevance Feedback)	68
One Index Over Many Fields (Integrated Metadata Search)	73
Merging the Search Engines (Rank Fusion)	78
Splitting the Search (Parallel Partitioned Search)	83
Same Person, Different Spelling (Record Matching and Linkage)	88

PREFACE: THE MACHINERY UNDER THE QUESTION

Every time you type something into a search box, tap a name in your contacts, or watch a page of results appear the instant you ask for them, a piece of forty-year-old genius is doing the work. You do not see it. That is the point of it. A database's whole job is to make finding one record among a billion feel like no work at all — and the algorithms that pull off that trick are among the most quietly brilliant inventions in computing.

They also have a history, and that history has receipts.

Between 1993 and 2001, the companies that ran the world's data — IBM, Oracle, Sybase, Informix, Tandem, Microsoft — filed patents on exactly how they made databases fast. Not vague ideas: specific, claimed machinery. How to lay out an index so it never has to scan. How to answer "how many, grouped by region" without touching the

rows. How to zip two sorted tables together in a single pass. How to store a set of a billion things as a stripe of bits and answer questions about it with a single machine instruction. Each of those filings was a twenty-year monopoly, granted in exchange for publishing, in exhaustive legal detail, precisely how the invention worked.

Twenty years have passed. Every patent in this book has expired. The monopolies are over; the knowledge is in the public domain — legally free for anyone to read, build, and ship. What generic drugs did for medicine and expired 3-D-printing patents did for desktop manufacturing, this series does for software. This volume is the database chapter of that inheritance.

There is one honesty this book keeps carefully. Many of these techniques are older than their patents. The balanced tree that Chapter 1 is built on was described by Rudolf Bayer and Edward McCreight in 1972; sorting and hashing are older than the companies that filed on them. What the patents of the late 1990s claimed was not the idea — it was the industrial machinery of applying it: a particular way to keep an index balanced under heavy updates, a particular trick to lock one value's bitmap without freezing the whole index, a specific method to prune the partitions a query can safely skip. So each chapter anchors to a real, datable patent — assignee, filing year, the claim in plain English — and where the core idea is older, *The Story* says so plainly. This is history with receipts, not a myth of lone invention.

And every line of code here runs. No Postgres, no Oracle, no cloud, no pip install beyond this one library — which itself has zero dependencies. Each algorithm is a single readable file you can run on a locked-down school laptop, a Raspberry Pi, or a five-pound board. That constraint is the ethos: the machinery that made billion-row databases fast is small enough to hold in your hand and read in an afternoon.

HOW TO READ THIS BOOK

No mathematics beyond high-school algebra. No database installed, no server running, no prior knowledge of SQL assumed — where a SQL idea matters, it is explained from zero. Install the companion library:

```
pip install patentml
```

Open a file, run it, change it, break it. Every chapter names the exact module it draws from, and every number printed in these pages came from a real run of that module's built-in demo. When the text says a query returned ['d1', 'd3'], that is what the code returned when this chapter was written. Run it yourself and you will get the same thing.

Read the chapters in order or dip in anywhere — each is self-contained. The book climbs from the single most important

structure in all of data management (the index) through bitmaps, joins, spatial data, storage, and finally to the algorithms that turn stored data into recommendations.

A NOTE ON PATENTS AND THE PUBLIC DOMAIN

A patent is a bargain society makes with an inventor: publish exactly how your invention works, in enough detail that a skilled person could rebuild it, and in return you get twenty years to be the only one allowed to. When the twenty years end, the second half of the bargain comes due — the knowledge becomes everyone's. That is not a loophole. It is the entire purpose of the system: temporary monopoly, permanent public knowledge.

The patents in this book have all reached that second half. Their expiry dates — verified against the filing record — have passed. The implementations printed here are the author's own code, MIT-licensed, free for you to use in anything. The ideas belong to everyone now.

This is a map, not legal advice. If you plan to build a commercial

product on a specific technique, confirm the status of the exact patent yourself — patent term depends on filing date and maintenance, and this book's job is to hand you the ideas and the working code, not to be your lawyer.

Now: how do you find one record among a billion without looking at the other billion-minus-one? That is where every database begins, and it is where we begin too.

PART I — THE INDEX: FINDING WITHOUT LOOKING

A database with no index is a filing cabinet with the drawers welded open and the folders thrown in at random. To find one thing you take everything out and look at all of it. That is a "full scan," and for a billion rows it is a catastrophe. An index is the alternative: a second, smaller structure whose only job is to tell you where to look so you never have to look everywhere. Almost everything a database does quickly, it does quickly because of an index. This first part of the book is the story of the index in its major forms — the balanced tree, the ordered shelf, the multi-column key, the word map, and the self-growing hash table. We start with the structure that, more than any other, deserves to be called the shape of stored data.

CHAPTER 1

THE BALANCED TREE (B-TREE INDEXES)

THE FILING

US Patent 5,832,500 — "Method for searching an index."

US Patent 5,966,710 — "Method for searching an index."

US Patent 5,852,820 — "Method for optimizing entries for searching an index."

Filed 1996–1998. All expired; public domain.

These three filings, granted to the database vendors of the late 1990s, read like a set of engineering notes on a single obsession: how to keep a tree-shaped index fast when millions of rows are being inserted, deleted, and searched all at once. The claims describe a multi-level structure of index nodes, each holding a sorted run of keys and pointers to child nodes, arranged so that a search "descends" from a single root down to the exact entry it wants —

comparing a handful of keys at each level rather than scanning them all. The "optimizing entries" patent is the telling one: it is not about building the tree but about keeping it balanced as it changes, so that no branch grows long and slow while another stays short. That balance is the whole game.

THE IDEA

Imagine your keys sorted on a shelf: 5, 6, 7, 10, 12, 17, 20, 30. To find 17, you could start at the left and walk right until you hit it. That is a scan, and it takes, on average, half the shelf.

Now do what you do with a physical dictionary instead. Open to the middle. Is 17 before or after what you see? After. Ignore the entire first half. Open to the middle of what remains. Repeat. Each look throws away half of what is left, so a shelf of a billion keys is searched in about thirty looks, not five hundred million. That is binary search, and it is the seed of the idea.

A B-tree is binary search made into a permanent, updatable structure. Instead of one long sorted shelf, you keep a shallow tree of small sorted shelves ("nodes"). Each node holds a few keys and, between and around them, pointers to child nodes holding the keys that fall in those gaps. To search, you start at the root node, find which gap your key falls into, follow that pointer down, and repeat. Because each node fans out to many children, the tree stays extremely shallow —

even a billion keys sit only four or five levels deep — so a search touches only four or five nodes.

The hard part is not searching. It is inserting. Drop a new key into a node and eventually that node overflows. A B-tree handles this with a single elegant rule: when a node is full, split it in two and push its middle key up to the parent. Splits ripple upward, and if the root itself splits, the tree gains a level — evenly, everywhere at once. That is why a B-tree can never become lopsided. It does not rebalance after the fact; it stays balanced by construction, on every single insert. That property — guaranteed shallow depth no matter the order things arrive in — is exactly what those "optimizing entries" claims were protecting.

THE STORY

The B-tree predates these patents by a quarter of a century. Rudolf Bayer and Edward McCreight described it in 1972 at Boeing Research Labs, and the "B" has been argued over ever since — Bayer, Boeing, balanced, broad; McCreight himself once said the more you think about what the B means, the better you understand the tree. It is one of the most important data structures ever invented, and by the 1990s it was the beating heart of every serious database and file system on earth.

So what were database companies patenting in 1996? Not the tree —

the industrialisation of it. By then a production database was not a textbook B-tree; it was a B-tree under savage real-world stress: thousands of concurrent transactions, keys arriving in every possible order, nodes on spinning disks where every level you avoid touching saves milliseconds that add up to fortunes. The patents of that era stake claims on the practical apparatus — the exact split strategy, the way entries are packed and searched within a node, the balancing discipline that keeps the worst case from ever arriving. These were the engineers who took a 1972 idea and made it survive a billion-row bank. The patent record is where that industrialisation left its fingerprints.

What did it become? Everything. The primary key of every SQL table, the file allocation table of your operating system, the index on this book's companion database — all B-trees or their close cousins (B+ trees, which chain the leaves for fast range scans). When you ask a database for "every order between March and June," a B-tree is what lets it jump to March and walk to June instead of reading the year.

THE CODE

Here is the search and the split from `patentml's btree_index` — the two operations that define the structure. Read insert against the rule above: fill a node, and when it is full, split it and push the middle key up.

```
def search(self, key):
    node = self.root
    while node:
        i = bisect.bisect_left(node.keys, key)
        if i < len(node.keys) and node.keys[i] == key:
            return node.vals[i]
        if node.leaf:
            return None
        node = node.children[i]    # descend into the right gap
    return None

def _split_child(self, parent, i):
    t = self.t
    y = parent.children[i]
    z = _Node(leaf=y.leaf)
    parent.keys.insert(i, y.keys[t - 1])    # middle key rises to the
parent
    parent.vals.insert(i, y.vals[t - 1])
    z.keys = y.keys[t:]; y.keys = y.keys[:t - 1]    # right half becomes a
new node
    z.vals = y.vals[t:]; y.vals = y.vals[:t - 1]
    if not y.leaf:
        z.children = y.children[t:]; y.children = y.children[:t]
    parent.children.insert(i + 1, z)
```

The module's built-in demo builds a small tree, then searches and

range-scans it. Running it prints:

B-tree index (from US5832500/US5966710/US5852820, expired):

```
search(6) -> v6
```

```
search(99) -> None
```

```
in-order -> [5, 6, 7, 10, 12, 17, 20, 30]
```

```
range 6..17 -> [6, 7, 10, 12, 17]
```

Three things worth noticing in that output. `search(6)` found its value by descending, not scanning. `search(99)` returned `None` after touching only the nodes on one root-to-leaf path — it proved a key's absence without reading the whole index. And `range 6..17` returned the keys in sorted order, because an in-order walk of a B-tree is sorted — the reason databases lean on B-trees for `ORDER BY` and `BETWEEN` as much as for point lookups.

The demo does one more thing you should trust: it builds two thousand random trees with different node sizes and checks every search against a plain Python dictionary and every traversal against a sorted list. It reports `PASS`. The structure is not just plausible; it is verified.

THE INHERITANCE

Now that these patents are expired, the balanced tree is yours for

nothing — and it is still, in 2026, the right answer astonishingly often. Reach for a B-tree whenever you need three things at once: fast lookup of a single key, fast scanning of a range of keys in order, and cheap updates as data changes. Hash tables (Chapter 5) beat it for pure point lookups but cannot do ranges at all. Sorted arrays (Chapter 2) match its search speed but choke on inserts. The B-tree is the structure that refuses to be bad at anything, and that all-round competence is why, half a century after Bayer and McCreight and a quarter-century after the patents that industrialised it, the shape of stored data is still a tree that keeps itself balanced.

CHAPTER 2

THE ORDERED SHELF (SORT-ORDER INDEXES)

THE FILING

US Patent 5,878,410 — "File system sort order indexes."
Filed 1996. Expired; public domain.

Where Chapter 1's patents guarded a tree, this one guards something humbler and, for many jobs, faster: a plain list kept permanently in sorted order. The claims describe maintaining an auxiliary index of records ordered by a sort key, so that a file system can present a sorted view — a directory listing, a range of dates, everything "starting with A" — by binary-searching into the ordered index instead of scanning every record and sorting it fresh each time. It is the difference between keeping your bookshelf alphabetised and re-alphabetising it every time someone asks for a title.

THE IDEA

A sort-order index is the simplest index that beats a scan: a list of keys held in sorted order, with each key pointing back to its record. Because it is sorted, three questions become cheap.

Lookup is binary search — the dictionary trick from Chapter 1 — landing on any key in about thirty comparisons per billion. Range ("everything from apple to banana") is two binary searches, one for each end, and then you simply read the slice between them; the answer is already contiguous and already in order. Prefix ("everything starting with ap") is the same move: binary-search to where ap would begin, then walk forward, emitting keys, and stop the instant one no longer starts with ap. You never look left of the start or right of the end.

The cost lives in one place: insertion. To add a key you must find its sorted position and shift the rest along to make room. For a structure that changes constantly, that shifting is why Chapter 1's tree exists. But for data that is read far more than it is written — a directory, a lookup table, a reference list built once and queried forever — the ordered shelf is often the fastest and simplest thing there is.

THE STORY

Keeping data sorted for fast lookup is as old as record-keeping;

librarians did it centuries before computers. What US5878410 captures is the moment the file system itself took on the job — maintaining sorted views as a first-class service so that applications above it got ordered listings and range queries for free. The late-1990s context was a world of exploding file counts and users who expected a folder of ten thousand items to sort instantly by name, date, or size. The patent's quiet insight is that you do not sort on demand; you keep a small sorted index up to date as files come and go, and every sorted question afterwards is a binary search.

Its descendants are everywhere you see something ordered without a visible delay: the sorted column in a spreadsheet, the "sort by price" on a shopping page, the ORDER BY that a database answers from an existing index rather than sorting the result set. The idea is so ordinary now that it is invisible — which is the highest compliment an algorithm can earn.

THE CODE

The whole structure is a sorted key list and a parallel record list, kept aligned with Python's `bisect`. Here is insertion and the prefix scan from `patentml's sort_order_index`:

```
def add(self, key, payload=None):
    i = bisect.bisect_right(self._keys, key)    # find sorted position
    self._keys.insert(i, key)                  # shift to make room
```

```
self._rows.insert(i, (key, payload))
return self

def prefix(self, pre):
    i = bisect.bisect_left(self._keys, pre)    # jump to where pre
    begins
    out = []
    for k, p in self._rows[i:]:
        if isinstance(k, str) and k.startswith(pre):
            out.append((k, p))
        else:
            break    # stop at the first miss
    return out
```

The demo inserts five fruit names in scrambled order, then asks three ordered questions. It prints:

Sort-order index (from US5878410, expired):

```
ordered    -> ['apple', 'apricot', 'avocado', 'banana', 'cherry']
prefix 'ap' -> ['apple', 'apricot']
range a..b  -> ['apple', 'apricot', 'avocado']
```

The names went in as banana, apple, apricot, cherry, avocado and came out alphabetised, because insertion places each one at its sorted position immediately. `prefix('ap')` returned exactly the two "ap"

words and stopped — it never inspected banana or cherry. And range a..b returned a contiguous slice with no sorting step at all. The demo then fuzz-tests four thousand random datasets, comparing its ordered output and range queries against Python's own sorted, and reports PASS.

THE INHERITANCE

The ordered shelf is the tool to reach for when your data is read far more than written and you want ordered lookups, ranges, and prefixes with almost no code. It is the read-optimised sibling of Chapter 1's tree: same search speed, simpler to build, but punishing on heavy inserts. Knowing which to pick — tree for churn, sorted list for stable reference data — is a small piece of judgement that separates a fast system from a slow one, and it costs you nothing now that both patents have expired. Keep this chapter's structure in your pocket; it is the answer more often than beginners expect.

CHAPTER 3

MANY KEYS AT ONCE (COMPOSITE INDEXES)

THE FILING

US Patent 5,918,225 — "SQL-based database system with improved indexing methodology."

Filed 1993. Expired; public domain.

One of the older filings in this book, and one of the most consequential, because it concerns the index that spans more than one column. The claims describe indexing rows on a concatenated multi-column key held in sorted order, so that a query fixing the leading column and ranging over the next resolves to a single contiguous slice of the index. Anyone who has ever wondered why a database index on (country, city) speeds up a search by country but not a search by city alone is looking at the rule this patent formalised.

THE IDEA

A composite index sorts rows by a tuple of columns, in order — first by the first column, and only within ties by the second, and so on. Think of a phone book, which is sorted by surname and then, within each surname, by first name. That single ordering makes some questions trivial and others useless, and the pattern is worth burning into memory because it governs how every real database index behaves.

Ask for "everyone named Smith" and the phone book is perfect: all the Smiths sit together in one block you can jump to and read. Ask for "everyone named Smith, first name between Andrew and David" and it is still perfect: within the Smith block, first names are sorted, so that too is one contiguous slice. But ask for "everyone whose first name is Andrew," regardless of surname, and the phone book is no help at all — the Andrews are scattered across every surname from Adams to Young. The index is sorted by the leftmost column first, so it only accelerates queries that pin down the leftmost columns before ranging over the next. This is the leftmost-prefix rule, and it is the single most useful thing a working database engineer knows about indexes.

THE STORY

By 1993, SQL had won. Relational databases were the standard, and

their whole promise was that you could ask arbitrary questions across arbitrary columns. The uncomfortable truth underneath that promise was that you could not make every combination fast — an index imposes one ordering, and one ordering privileges some questions over others. US5918225 is part of the industry working out how to expose that reality usefully: let the database designer declare composite indexes on the column-combinations that matter, and teach the query planner the leftmost-prefix rule so it knows when an index applies and when it must fall back to a scan.

The legacy is the daily craft of database tuning. Every time an engineer adds an index on (customer_id, order_date) because the app always filters by customer and then narrows by date — and pointedly does not add a separate index on order_date because no query needs it alone — they are applying this chapter. The composite index is why fast databases feel fast on the queries their designers anticipated, and mysteriously slow on the ones they did not.

THE CODE

The structure is a sorted list of tuple-keys, searched with sentinels that mark "smaller than anything" and "larger than anything" so a prefix expands to an exact slice. Here is the leftmost-prefix range query from patentml's composite_index:

```
def prefix_range(self, prefix, col_lo, col_hi):
```

```

""Equality on prefix, then a range on the next column.""
p = tuple(prefix)
rest = self.ncols - len(p) - 1
lo = bisect.bisect_left(self._keys, p + (col_lo,) + (_MIN,) * rest)
hi = bisect.bisect_right(self._keys, p + (col_hi,) + (_MAX,) * rest)
return self._rows[lo:hi]

```

The demo indexes rows keyed on (country, number) and asks two questions. It prints:

Composite index (from US5918225, expired):

```

eq_prefix ('uk',)      -> [('uk', 1), ('uk', 3), ('uk', 5)]
('uk',) then col 2..5  -> [('uk', 3), ('uk', 5)]

```

The first query fixed the leading column and got all three UK rows as one block — the "Smith" case. The second fixed the leading column and ranged over the second, and got a slice within that block — the "Smith, first name A–D" case. Both were answered by two binary searches into a single sorted structure, no scanning. The module fuzz-tests three thousand datasets against a brute-force scan and reports PASS, confirming the slice always equals what a full filter would find.

THE INHERITANCE

The composite index is free now, and the judgement that comes with it is the real inheritance: order your index columns by how your queries filter, most-fixed first. Put the column you always pin down on the left and the column you range over next. Get the order right and one index serves a whole family of queries; get it backwards and the index sits there, built and useless, while the database scans. No library gives you that instinct — this chapter does, and it applies identically whether the engine underneath is SQLite on a Raspberry Pi or a cluster the size of a warehouse.

CHAPTER 4

THE WORD MAP (THE INVERTED INDEX)

THE FILING

US Patent 6,021,409 — "Method for parsing, indexing and searching world-wide-web pages."

Filed 1996. Expired; public domain.

This is the patent behind the search box. Filed at the dawn of the web, its claims describe parsing text into indexable words and storing, for each unique word, a list of the locations — which document, and where within it — that the word appears. Searching then becomes a lookup of the word's list rather than a reading of every document. Every web search engine, every "find in your email," every site search you have ever used runs on the structure this filing describes.

THE IDEA

Turn the obvious arrangement inside out. The obvious way to store documents is document-by-document: here is document 1 and its words, here is document 2 and its words. To find which documents contain "brown," you would have to open and read all of them. That is a scan, across the whole corpus, for every search — hopeless at web scale.

The inverted index flips it. Instead of mapping documents to their words, map each word to the documents that contain it. Now "brown" is a single key; look it up and you get, instantly, the list of every document that used it — no reading required. That list of locations is called a postings list, and building the whole index is a one-time cost paid at ingestion so that every future search is a dictionary lookup.

Once each word points at its documents, the operations of search fall out for free. AND ("quick AND fox") is the intersection of two postings lists — documents that appear in both. OR is the union. And if you store not just which document but where in it each word sits, you can answer phrase queries ("quick brown" as consecutive words) by checking whether the positions line up: does "brown" appear at some position exactly one after "quick"? The humble idea of listing locations turns into Boolean search and phrase search with nothing added but bookkeeping.

THE STORY

The inverted index is older than the web — information-retrieval researchers used it on document collections for decades, and a concordance to the Bible is a hand-built inverted index made by monks. What changed in 1996 was scale and stakes. The web was exploding, nobody could read it, and whoever could index it would own the front door to the internet. US6021409 is a filing from exactly that scramble: parse the messy text of a web page, extract its words, record where each one lives, and make the whole ungovernable web answerable by a lookup. Within a few years the companies that industrialised this structure — pointing enormous postings lists at enormous crawls of the web — became the most valuable companies on earth. The inverted index is arguably the single most economically important data structure of the internet age.

THE CODE

Building it is almost anticlimactic: walk each document's words and, for each, append this document and position to that word's list. Here is the core of `patentml`'s `inverted_index`, including the phrase check that reads positions:

```
def add(self, doc_id, text):
    self.docs.add(doc_id)
    for pos, w in enumerate(tokenize(text)):
```

```
        self.postings[w].setdefault(doc_id, []).append(pos)
    return self

def phrase(self, text):
    ws = tokenize(text)
    out = []
    for d in self.search_and(*ws):          # docs with all the words
        pos_sets = [set(self.postings[w][d]) for w in ws]
        # is there a start position p where every word lands at p+i?
        if any(all((p + i) in pos_sets[i] for i in range(len(ws))))
            for p in pos_sets[0]):
            out.append(d)
    return out
```

The demo indexes three short documents and asks four questions. It prints:

Inverted index (from US6021409, expired):

```
search('brown')      -> ['d1', 'd2', 'd3']
search_and(quick,fox) -> ['d1']
search_or(lazy,jump)  -> ['d2', 'd3']
phrase('quick brown') -> ['d1', 'd3']
```

Read those four lines and you have read the whole of search.
"brown" appeared in all three documents, returned by a single

lookup. "quick AND fox" intersected two lists down to the one document with both. "lazy OR jump" took the union. And "quick brown" as a phrase returned only the documents where the two words sat next to each other — d1 (the quick brown fox) and d3 (quick brown foxes jump) — while correctly excluding any document that merely contained both words apart. The module fuzz-tests three thousand random document sets, checking search, AND, and phrase against brute-force scans, and reports PASS.

THE INHERITANCE

The inverted index is public domain, and it is still the right tool the moment you have text and questions about it. You do not need a search company or a heavyweight engine; a few hundred lines gives you real Boolean and phrase search over a personal archive, a codebase, a pile of PDFs, a chat log — on a laptop, offline, private. In an age where "search" is sold back to us as a cloud service that reads everything we own, there is something quietly radical about the fact that its foundational structure fits in a single file you can run on a five-pound board. The front door to the internet turns out to be a dictionary from words to lists. Now it is yours.

CHAPTER 5

THE TABLE THAT GROWS ITSELF (EXTENDIBLE HASHING)

THE FILING

US Patent 5,893,086 — "Parallel file system and method with extensible hashing."

Filed 1996. Expired; public domain.

The final index in Part I is the fastest for one specific job and the trickiest to grow. The claims describe a hash-based index whose directory expands on demand: a table of pointers indexed by the leading bits of a key's hash, where each storage bucket carries a "local depth," and when a bucket overflows it splits — doubling the directory only when it must. The prize is a lookup that stays constant-time no matter how much data you add, without ever stopping the world to rebuild the whole table.

THE IDEA

A hash table is the fastest lookup there is: run the key through a hash function to get a number, use that number to jump straight to a slot, done — one step, no searching, no matter how many keys. The catch is what happens when the table fills. The textbook fix is to build a bigger table and re-hash every existing key into it — a "stop-the-world" rehash that, on a large database, freezes everything for an agonising moment. Extendible hashing is the elegant escape from that freeze.

The trick is a layer of indirection: a directory of pointers. You index the directory using the top d bits of a key's hash. Each directory entry points to a bucket that physically holds keys. Crucially, several directory entries can point to the same bucket, and each bucket remembers its own "local depth" — how many hash bits actually distinguish its contents. When a bucket overflows, you split just that bucket into two, hand each half the keys according to their next hash bit, and repoint the directory. Only if the overflowing bucket was already as deep as the directory itself do you double the directory — and even then you touch no keys, just pointers. Growth is incremental and local. The table expands itself, one bucket at a time, and no lookup ever gets slower than a single step.

THE STORY

Hashing as an idea dates to the 1950s; extendible hashing specifically was published by Fagin, Nievergelt, Pippenger and Strong at IBM in 1979, as a way to keep hash-based files efficient as they grew on disk. US5893086's contribution, seventeen years later, is the setting: a parallel file system, where many nodes hammer the same index at once and a stop-the-world rehash is not merely slow but catastrophic — it would stall an entire cluster. The patent industrialises extendible hashing for that world, where incremental, local, non-blocking growth is not a nicety but a requirement. Its descendants live in the guts of distributed file systems and databases wherever a hash index must scale smoothly under constant concurrent pressure.

THE CODE

The heart is the split: grow the directory only when forced, then partition one bucket's keys by their next hash bit. Here is `_split` from `patentml`'s `extendible_hash`:

```
def _split(self, idx):
    b = self.dir[idx]
    if b.local_depth == self.global_depth:    # only now must the
directory double
        self.dir += self.dir
        self.global_depth += 1
    new_ld = b.local_depth + 1
```

```
b0, b1 = _Bucket(new_ld), _Bucket(new_ld)
high_bit = 1 << b.local_depth
for k, v in b.items.items():          # re-home ONLY this bucket's
keys
    (b1 if self._hash(k) & high_bit else b0).items[k] = v
for i in range(len(self.dir)):        # repoint the directory
    if self.dir[i] is b:
        self.dir[i] = b1 if i & high_bit else b0
```

The demo inserts ten keys into buckets that hold only two apiece, forcing several splits. It prints:

Extendible hashing (from US5893086, expired):

```
get(7)      -> 49
global_depth -> 3 (grew from 0 without rehashing existing keys)
total stored -> 10
```

That middle line is the whole point. The directory grew from depth 0 to depth 3 — doubling twice — to accommodate ten keys in two-slot buckets, yet no existing key was ever re-hashed into a from-scratch table. Each split moved only the keys in one overflowing bucket. `get(7)` still returns its value in a single step, and all ten keys are present and findable. The module fuzz-tests three thousand tables with varying bucket sizes, checking every lookup and the full contents against a plain Python dictionary, and reports PASS.

THE INHERITANCE

Extendible hashing is yours now, and it earns its place whenever you need the raw speed of a hash table but cannot tolerate the pause of a rehash — a long-running index that grows without warning, a shared table under concurrent load, anything where a freeze is unacceptable. It is the specialist of Part I: unbeatable for point lookups, useless for ranges (a hash scatters neighbouring keys to the winds, so there is no "next key" — that is what Chapters 1 and 2 are for). Knowing to reach for a self-growing hash when you need constant-time lookups and graceful growth, and for a tree when you need order, is exactly the fluency this part of the book was built to give you. With that, the index — the thing that lets a database find without looking — is in your hands in all its major forms.

PART II — BITMAPS: SETS AS BITS

The indexes of Part I answer "where is this key?" This second part answers a different and, for analytics, more valuable question: "which rows satisfy all of these conditions at once?" Its trick is to represent a set of rows not as a list but as a stripe of bits — one bit per row, on or off — so that combining conditions becomes bitwise arithmetic the processor does sixty-four rows at a time. It is the technique that made "how many customers in the north, aged over fifty, who bought in March" answerable in a blink over tables of billions. Two chapters: the bitmap index itself, and the compression that makes it fit.

CHAPTER 6

SETS AS BITS (BITMAP INDEXES)

THE FILING

US Patent 5,884,307 — "Updating bitmapped indexes."

US Patent 5,899,988 — "Bitmapped indexing with high granularity locking."

Both filed 28 February 1997. Both expired 28 February 2017; public domain.

Filed on the same day, this pair is the bitmap index in its industrial form. The first patent concerns keeping such an index correct as rows change; the second adds a concurrency trick — locking at the granularity of a single value's bitmap, so that two users updating different values do not block each other. Together they describe the machinery behind one of the signature features of the great analytic databases: an index that turns "rows matching this value" into a

stripe of bits, and turns combining conditions into the fastest operations a computer knows.

THE IDEA

Suppose a column holds a value from a small set — a colour that is red, blue, or green; a region; a yes/no flag. A bitmap index keeps, for each distinct value, a bitmap: a long row of bits, one per record, where the bit is 1 if that record has that value and 0 otherwise. The red bitmap for six rows coloured red, blue, red, green, blue, red is 1 0 1 0 0 1 — on at rows 0, 2, and 5.

Now watch what combining conditions becomes. "Red OR green?" Take the red bitmap and the green bitmap and OR them together, bit by bit. "NOT red?" Flip every bit of the red bitmap. "Blue-or-green AND NOT red?" OR the blue and green bitmaps, AND with the flipped red bitmap. Each of these is a bitwise operation — AND, OR, NOT — and a processor performs them on sixty-four bits, meaning sixty-four rows, in a single instruction. A query that would scan a billion rows becomes a handful of bitwise passes over compact bit-stripes. For columns with few distinct values, nothing else on earth is this fast.

The two patents add the realities. Bitmaps must be updated when a row's value changes — flip the bit off in the old value's bitmap, on in the new one — which is US5884307. And in a busy system, you do

not want to lock the entire index every time one row changes; US5899988's "high granularity locking" locks only the affected value's bitmap, so unrelated updates run in parallel. patentml's module models this by versioning each value's bitmap independently.

THE STORY

The bitmap index rose to fame with the analytic databases of the 1990s — Model 204 pioneered it years earlier, but it was the data-warehouse boom, and companies like Oracle building it into their engines, that made it famous. It solved a problem that ordinary tree indexes handle badly: columns with low cardinality, a handful of repeated values, where a B-tree is bloated and clumsy but a bitmap is tiny and lightning-fast. The classic example is exactly the analyst's bread and butter — "count the customers who are female, in the north, aged over fifty, and lapsed" — four low-cardinality conditions AND-ed together, which a bitmap index answers by AND-ing four bit-stripes and counting the 1s. This is the technique underneath the phrase "our data warehouse can slice a billion rows in real time," and these two 1997 filings are where the mainstream database industry staked its claim on the updatable, concurrent version of it.

THE CODE

Bitmaps here are Python's arbitrary-precision integers, so AND, OR,

and NOT are the language's own `&`, `|`, and `~` — machine-word-parallel for free. Here is the predicate and combination core of `patentml`'s `bitmap_index`:

```
def eq(self, value):          # the bitmap for one value
    return self.bitmaps.get(value, 0)

def any_of(self, values):     # OR of several values' bitmaps
    bm = 0
    for v in values:
        bm |= self.bitmaps.get(v, 0)
    return bm

def not_(self, bitmap):      # flip within the known rows
    return self._all() & ~bitmap

@staticmethod
def and_(*bitmaps):          # AND of any number of predicates
    out = bitmaps[0] if bitmaps else 0
    for bm in bitmaps[1:]:
        out &= bm
    return out
```

The demo indexes six coloured rows, then combines conditions. It prints:

Bitmap index (from US5884307 + US5899988, expired 2017):

```
rows      : ['red', 'blue', 'red', 'green', 'blue', 'red']
eq('red')  -> [0, 2, 5]
red OR green-> [0, 2, 3, 5]
NOT red    -> [1, 3, 4]
blue|green AND NOT red -> [1, 3, 4]
after set(0,'green'), eq('green') -> [0, 3]
```

Every line is a bitwise operation made readable. `eq('red')` is the red bitmap, on at 0, 2, 5. `red OR green` unions two stripes. `NOT red` flips them. The compound `blue|green AND NOT red` — a real analytic filter — resolves to rows 1, 3, 4 by OR-ing then AND-ing, no scan. And the last line shows US5884307's update at work: changing row 0 from red to green flips one bit in each of two bitmaps, and green now correctly reports rows 0 and 3. The module fuzz-tests four thousand columns, checking equality, AND, NOT, and point-updates against brute force, and reports PASS.

THE INHERITANCE

The bitmap index is public domain, and it remains the champion for one shape of problem: many conditions, each on a low-cardinality column, combined and counted over a large, mostly-read table. That is the shape of analytics — dashboards, cohort counts, "how many match all of these." Reach for a bitmap index there and you get

answers at the speed of bitwise arithmetic; reach for it on a high-churn transactional table with millions of distinct values and it becomes a liability. The judgement — bitmaps for wide analytic filters on few-valued columns, trees for everything ordered and unique — is the working knowledge these two same-day 1997 patents encode, now free for the taking. The next chapter answers the obvious objection: a bitmap with a billion rows is a billion bits per value. How do you keep that from becoming enormous?

CHAPTER 7

SQUEEZING THE BITS (COMPRESSED BITMAPS)

THE FILING

US Patent 6,141,656 — "Query processing using compressed bitmaps."

Filed 1998. Expired; public domain.

Chapter 6 ended on a worry: a bitmap for a billion-row table is a billion bits per value, and most of those bits are 0. This patent is the answer. Its claims describe keeping the bitmap index run-length compressed — long stretches of 0s or 1s stored as a count rather than as the bits themselves — and, critically, evaluating the Boolean queries directly on the compressed form, never expanding it back to full length. You get the bitmap index's speed at a fraction of its size, because the compression and the query are the same pass.

THE IDEA

Bitmaps are wildly compressible because they are mostly one value. A bitmap that is on at a few rows and off everywhere else is a vast desert of 0s with a few oases of 1s. Storing a billion literal bits to represent a few thousand set rows is absurd — so you store the runs instead: "off for the next million, then on for three, then off for the next two million." That is run-length encoding, and on the sparse bitmaps that fill a real index it shrinks the data by orders of magnitude.

But compression that you have to undo before you can use it buys you nothing at query time — you would spend the query re-inflating billion-bit stripes. US6141656's insight is that the Boolean operations can be performed on the runs directly. `patentml`'s module represents each bitmap as a sorted list of [start, end) intervals of set bits, and then:

- OR of two bitmaps is the merge of their intervals — lay all the runs on a line and coalesce overlaps.
- AND is the overlap of their intervals — walk both interval lists together and emit only where they cover the same rows.
- NOT is the gaps between the intervals, up to the table size.

None of these ever materialises a full bit-stripe. A query over a billion-row table touches only as many runs as the data actually has

— which, for the sparse bitmaps that dominate an index, is tiny. The index is small on disk and fast to query, because being small and being fast are, here, the same property.

THE STORY

This is the family of techniques — Byte-aligned Bitmap Compression, Word-Aligned Hybrid, and their relatives — that made bitmap indexes practical at scale rather than merely elegant. The uncompressed bitmap index of Chapter 6 is beautiful on paper and ruinous in storage the moment cardinality creeps up or tables grow huge. The compressed bitmap is what shipped. US6141656 sits in the late-1990s wave of data-warehouse engineering where the goal was to make "slice a billion rows on ten conditions" not just possible but affordable, on the disks of the day. Its descendants are in every column-store and analytic engine that boasts bitmap or "roaring" indexes today; the modern Roaring Bitmap format that underpins a generation of analytic systems is the direct intellectual heir of this run-and-block-compressed lineage.

THE CODE

Boolean operations become interval algebra. Here are the AND (interval overlap) and OR (interval merge) from `patentml`'s `compressed_bitmap`, operating on runs, never on bits:

```

def _intersect(a, b):          # AND: walk both, emit overlaps
    out = []; i = j = 0
    while i < len(a) and j < len(b):
        s = max(a[i][0], b[j][0]); e = min(a[i][1], b[j][1])
        if s < e:
            out.append((s, e))
        if a[i][1] < b[j][1]: i += 1
        else:                  j += 1
    return out

def _union(a, b):             # OR: sort all runs, coalesce overlaps
    out = []
    for s, e in sorted(a + b):
        if out and s <= out[-1][1]:
            out[-1] = (out[-1][0], max(out[-1][1], e))
        else:
            out.append((s, e))
    return out

```

The demo builds two compressed bitmaps and combines them. It prints:

Compressed bitmap (from US6141656, expired):

```

A      : [0, 1, 2, 3, 8, 9] (stored in 2 runs, not 10 bits)
A AND B : [2, 3, 9]

```

A OR B : [0, 1, 2, 3, 4, 8, 9, 10]

NOT A (u=11) : [4, 5, 6, 7, 10]

Look at the first line: the set {0,1,2,3,8,9} is stored as two runs — [0,4) and [8,10) — not ten bits. On a real index that is the difference between gigabytes and megabytes. Then A AND B, A OR B, and NOT A all produce exactly the right rows, computed by merging and overlapping intervals without ever expanding to a bit-stripe. The module fuzz-tests five thousand random sets, checking AND, OR, NOT, and membership against Python's own set operations, and reports PASS — the compressed form gives byte-identical answers to the uncompressed one.

THE INHERITANCE

The compressed bitmap is the version of Chapter 6 you would actually ship, and it is now entirely free. It is the tool for sparse sets over huge universes — row-sets in an analytic index, the classic use, but also any place you find yourself holding "which of these millions of things are flagged": a permission matrix, a feature-membership set, a search-result set to be intersected across facets. Store it as runs and your set is small on disk and fast to combine, because compression and query became one operation. With Parts I and II behind you, you now hold both halves of the database's core: the indexes that find a key, and the bitmaps that combine conditions across billions of rows. Part III turns from finding rows to answering

the questions we actually ask of them — the joins, the aggregates, and the queries.

PART III — THE QUERY: ANSWERING FAST

Parts I and II built the structures that let a database find rows. But finding rows is not what a user asks for. A user asks a question — "which customers placed orders," "how much did each region sell," "how many errors in the last minute," "who is this, despite the typo in their name." Answering those questions fast is a craft of its own, and it is the craft that separates a database from a mere pile of indexed data. This part is the query in its essential forms: joining two tables, keeping answers pre-computed, asking questions of a moving stream, refining a vague search, filtering across many fields, merging results from many engines, splitting work across machines, and matching records that should be the same but are spelled differently. Eight chapters, each a way of turning stored data into an answer.

CHAPTER 8

ZIPPING TWO TABLES TOGETHER (THE SORT-MERGE JOIN)

THE FILING

US Patent 6,185,557 — "Merge join process."

Filed 31 July 1998. Expired 31 July 2018; public domain.

The join is the operation that makes a relational database relational: it connects rows in one table to related rows in another — customers to their orders, employees to their departments, this to that. This patent concerns one of the two or three fundamental ways to do it. Its claims describe sorting both inputs on the join key and then advancing two cursors through them together, emitting matches as the keys line up — a single linear pass after the sort, handling duplicate keys on both sides correctly. It is the join you reach for when the inputs are large and already sorted or indexed.

THE IDEA

You have two tables and you want to pair up rows that share a key — say, users and orders, joined on `user_id`. The naive way is a nested loop: for every user, scan every order looking for a match. For a thousand users and a thousand orders that is a million comparisons; for a million of each it is a trillion, and your query never returns.

The sort-merge join does something a card-sorter would recognise. First, sort both piles by the join key. Now walk two fingers down the piles at once. If the left key is smaller than the right, the left finger has no possible match ahead of it — advance it. If the right is smaller, advance the right. And when the two keys are equal, you have found a match: emit it, and because there may be several equal keys on each side, emit the full cross-product of the equal runs — every left row with key `k` paired to every right row with key `k` — before moving both fingers past them. After the sort, the merge is a single pass down each table. A trillion-comparison catastrophe becomes a walk.

The duplicate handling is the subtle part the patent is careful about. Two users can share an id in a messy dataset; a user can have many orders. When both sides have a run of equal keys, the correct answer is every pairing between them, and the merge must emit that block before advancing. Get that right and the sort-merge join gives exactly the same rows as the nested loop, in a fraction of the time.

THE STORY

The join algorithms — nested-loop, sort-merge, and hash join — are foundational database theory worked out in the 1970s and 80s as the relational model took hold. Sort-merge is the classic for large inputs, especially when the data is already sorted, because then you skip the sort entirely and the whole join is one linear pass. US6185557, filed in 1998, is the industrial refinement of that classic inside a production query engine — the details of driving the two cursors, handling the equal runs, integrating with the sort the database was often doing anyway. Its lineage runs straight into every SQL engine's query planner, which to this day chooses between nested-loop, sort-merge, and hash join based on the size and sortedness of the inputs. When you read a database's "explain plan" and see the words MERGE JOIN, this chapter is what it means.

THE CODE

The whole method is the two-cursor walk with cross-product on equal runs. Here is the heart of patentml's `merge_join`:

```
while i < n and j < m:
    a, b = lk(L[i]), rk(R[j])
    if a < b:
        i += 1          # left key can't match ahead; advance left
    elif a > b:
```

```
j += 1                # advance right
else:
    i0 = i
    while i < n and lk(L[i]) == a: # the equal run on the left
        i += 1
    j0 = j
    while j < m and rk(R[j]) == b: # the equal run on the right
        j += 1
    for li in range(i0, i):        # emit the full cross-product
        for rj in range(j0, j):
            out.append((L[li], R[rj]))
```

The demo joins a users table (two rows share id 2) to an orders table (id 2 has two orders). It prints:

Sort-merge join (from US6185557, expired 2018):

```
(2, 'bob') x (2, 'book')
(2, 'bob') x (2, 'pen')
(2, 'bea') x (2, 'book')
(2, 'bea') x (2, 'pen')
(4, 'dan') x (4, 'hat')
```

That output is the duplicate handling made visible. Key 2 has two users (bob, bea) and two orders (book, pen), so the correct join emits all four pairings — the cross-product of the equal runs — which is

exactly what appears. Key 4 has one of each, so one row. Keys with no match on the other side (a user with no orders, an order with no user) produce nothing, as an inner join should. The module fuzz-tests five thousand random relation pairs, including duplicate keys, against a brute-force nested-loop join, and reports PASS.

THE INHERITANCE

The sort-merge join is public domain, and it remains the right join for a specific and common situation: two large inputs, especially ones that arrive already sorted on the join key — which, if you indexed them (Part I), they effectively are. Its cousin the hash join wins when one side is small enough to fit in memory; the nested loop wins only for tiny inputs. Knowing which join fits which shape of data is a large part of why one engineer's query returns in a second and another's runs all night over the same tables. That judgement — and a correct, duplicate-safe implementation to build on — is yours now, twenty years after the filing that industrialised it.

CHAPTER 9

THE ANSWER KEPT READY (AGGREGATE INDEXES)

THE FILING

US Patent 6,487,546 — "Apparatus and method for aggregate indexes."

Filed 27 August 1998. Expired 27 August 2018; public domain.

Some questions are asked constantly: total sales by region, order count by customer, average rating by product. Recomputing them from scratch every time — scanning and grouping millions of rows — is wasteful when the answer barely changes between asks. This patent's claims describe maintaining the grouped aggregates incrementally as rows are inserted, so that a "group by" query becomes a lookup of an already-computed answer rather than a scan. It is the pre-computed summary, kept correct on every insert.

THE IDEA

A GROUP BY query — "total sales, grouped by region" — normally works by scanning every row, sorting or hashing them into groups, and summing each group. Do that for a dashboard that refreshes every few seconds over a table of a hundred million rows and you are burning enormous effort to recompute an answer that changed by one sale.

The aggregate index keeps the answer ready instead. For each group, it maintains running statistics — count, sum, minimum, maximum — and updates them as each new row arrives: bump the group's count, add to its sum, widen its min/max if needed. The work is spread out, a tiny constant per insert, so that the query itself is free: to get "total sales in the north," you look up the north group's running sum. There is no scan, no grouping, no sort. The aggregate was computed once, incrementally, and simply read back.

The elegance is that the four classic aggregates — count, sum, min, max — are all incrementally maintainable: you can update each of them knowing only the new value and the old running total, never needing to revisit the rows already seen. Average falls out for free as sum divided by count. This is why the technique works at all: not every aggregate can be maintained this way (a median cannot, without keeping all the values), but the common ones can, and those are the ones dashboards ask for.

THE STORY

This is the database engine's version of a spreadsheet's running total, industrialised. The idea belongs to the family of materialized views and summary tables — pre-computed, stored answers that trade a little update cost for a lot of query speed — that became central to data warehousing in the 1990s. US6487546 is a filing from that wave, concerned specifically with keeping the aggregate index correct on insert without a rebuild, which is the hard and valuable part: a stale summary is worse than none. Its descendants are everywhere analytics must feel instant: the pre-aggregated rollups in OLAP cubes, the incremental materialized views in modern warehouses, the counters behind every "1,204 likes" that updates without recounting every like. When a billion-row dashboard answers in a blink, an aggregate index is usually why.

THE CODE

The whole technique is the incremental update on insert. Here is add from patentml's `aggregate_index`:

```
def add(self, row):
    g = self.group_key(row)
    v = self.value(row)
    s = self.stats.get(g)
    if s is None:
```

```
self.stats[g] = {"count": 1, "sum": v, "min": v, "max": v}
else:
    s["count"] += 1
    s["sum"] += v
    if v < s["min"]: s["min"] = v
    if v > s["max"]: s["max"] = v
return self
```

The demo feeds five sales rows tagged by region and then reads the rollups. It prints:

Aggregate index (from US6487546, expired 2018):

```
north count=2 sum=160 min=60 max=100 avg=80.0
south count=2 sum=50 min=10 max=40 avg=25.0
west count=1 sum=25 min=25 max=25 avg=25.0
```

Every one of those numbers was computed as the rows arrived, not by a scan at query time. The north region's sum=160 grew by 100 then 60 as its two rows landed; its min and max settled at 60 and 100; its average is simply 160 divided by 2. Asking the question after the fact costs a single dictionary lookup per group. The module fuzz-tests four thousand random tables, checking count, sum, min, and max for every group against a full GROUP BY scan, and reports PASS — the incremental answer always equals the from-scratch one.

THE INHERITANCE

The aggregate index is free now, and it is the tool for any dashboard, counter, or report where the same grouped question is asked far more often than the data changes. Maintain the rollup on write and reads become instant; the cost is a few arithmetic operations per insert, which is nothing beside the scans you avoid. The one thing to remember is its boundary — it maintains only the distributive aggregates (count, sum, min, max, and average from them). The moment you need a median, a distinct-count, or a percentile, you are back to keeping more state, which is a later chapter's problem. For the questions dashboards actually ask, the answer kept ready is one of the highest-leverage tricks in databases, and it is yours.

CHAPTER 10

QUESTIONS ON A MOVING RIVER (EVENT FILTERING AND WINDOWED AGGREGATION)

THE FILING

US Patent 6,367,034 — "Using query language for event filtering and aggregation."

Filed 1998. Expired; public domain.

Every chapter so far has queried data that sits still. This one queries data that flows — a never-ending stream of events, where the interesting question is not "what is in the table" but "what is happening right now." The claims describe expressing, in a query-like form, which events to keep (a filter) and how to aggregate the survivors over a sliding time window grouped by a key. Events pour past; the system keeps a running answer over the recent past and lets the ancient past fall away. It is the ancestor of streaming SQL.

THE IDEA

Imagine standing at a river and being asked, continuously, "how many red leaves floated past in the last minute?" You cannot store the whole river. You can only watch what passes and remember a recent window of it. That is stream processing, and it inverts the usual database assumption: the data does not wait for your query; your query waits for the data.

Two operations make it work. First, a filter: as each event arrives, a predicate decides whether it matters — keep the error logs, ignore the routine ones; keep the red leaves, ignore the brown. Most events are discarded on arrival, so you never store them. Second, a windowed aggregation: the events that survive the filter are folded into running totals, grouped by some key (per service, per user, per sensor), but only over a recent time window. As the clock advances, events older than the window are evicted — dropped from the totals — so the answer always reflects the last minute, the last hour, the last thousand events, and nothing staler.

The subtlety, and the one this book's implementation had to get exactly right, is the eviction clock. The window must advance on every event's timestamp, not only on the ones that survive the filter — otherwise a lull in interesting events lets the window freeze and stale data lingers in the totals. The answer to a question on a moving river is only ever as fresh as its oldest un-evicted event.

THE STORY

Databases spent their first decades assuming data was at rest — stored, then queried. By the late 1990s, whole domains had appeared where that assumption failed: network monitoring, financial tick data, telemetry, fraud detection, all producing torrents of events where the value was in the now and storing everything first was impossible. US6367034 sits at that turn, applying the familiar comfort of a query language — the SQL-shaped "select these, group by that, aggregate the rest" — to data in motion rather than at rest. That marriage of declarative queries with streaming data became a field of its own: Complex Event Processing, and then the streaming-SQL engines that run today's real-time dashboards, alerting systems, and fraud monitors. Every "errors per service in the last five minutes" alert is a direct descendant of the idea in this filing.

THE CODE

The eviction discipline is the crux — advance the clock on every event, keep only what falls inside the window. Here is push from patentml's `event_filter`:

```
def push(self, event, ts):
    self._evict(ts)          # advance the clock on EVERY event
    if not self.pred(event):  # filter: drop what doesn't matter
        return
```

```
self._events.append((ts, self.gkey(event), self.value(event)))

def _evict(self, now):
    if self.window is None:
        return
    while self._events and self._events[0][0] <= now - self.window:
        self._events.popleft()    # drop events older than the window
```

The demo streams six log events (a mix of error and info across two services) and asks for error counts per service in the last ten seconds, evaluated at time 20. It prints:

Event filter & aggregation (from US6367034, expired):

```
errors per service in last 10s (now=20):
    auth  count=1
    pay   count=1
```

Read that against the stream. auth logged errors at times 1 and 8 and again at 20; at "now = 20" with a ten-second window, only the error at time 20 is still inside it — count 1. pay errored at 5 and 13; only 13 survives — count 1. The early errors were correctly evicted as the clock advanced, and the info events were filtered out on arrival and never counted at all. The module fuzz-tests three thousand random streams, comparing its windowed counts and sums against a recomputed reference, and reports PASS.

THE INHERITANCE

Streaming aggregation is public domain, and it is the tool for any question about the recent present rather than the stored past: live dashboards, rate alerts ("more than 100 errors a minute"), sliding-window metrics, simple fraud and anomaly triggers. You do not need a heavyweight streaming platform to start — a filter, a time-ordered buffer, and a disciplined eviction clock give you real windowed analytics in a single file, on a device as small as a sensor's gateway. In a world drowning in event streams, the ability to ask a standing question of a moving river, and to answer it without storing the river, is one of the most quietly powerful ideas in data — and it has been free since this patent's term ran out.

CHAPTER 11

ASKING A BETTER QUESTION (QUERY REFINEMENT AND RELEVANCE FEEDBACK)

THE FILING

US Patent 5,987,457 — "Query refinement method for searching documents."

Filed 1997. Expired; public domain.

The search of Chapter 4 assumed you knew the right words. Often you do not — you type something vague, get a mix of hits, and only then discover what you meant by noticing which results are right and which are wrong. This patent turns that noticing into a better query automatically. Its claims describe refining a search by pushing the query toward the documents the user marked relevant and away from the ones marked irrelevant, surfacing new discriminating terms the user never typed. It is the search box that learns from your reaction to its own results.

THE IDEA

The classic problem: you search jaguar. Half the results are about the car, half about the animal. You wanted the car. Rather than making you think of better words, the system watches which results you click (or asks you to mark a few) and rewrites your query for you.

The method, called Rocchio relevance feedback, treats every document and the query itself as a bag of weighted words — a vector. It then moves the query vector in the space of words: add a share of the average of the documents you liked, subtract a share of the average of the ones you did not. Terms common in the good results get heavier; terms common in the bad results get lighter or vanish. The refined query is no longer just jaguar — it has grown terms like car, luxury, engine, pulled in from the relevant documents, and shed terms like jungle and cat that marked the irrelevant ones. You never typed those words. The feedback loop discovered them.

The formula has three knobs — how much to trust the original query, the relevant set, and the irrelevant set — and one safeguard: term weights that go negative are clamped to zero, because a query cannot contain "minus jungle," only "not weighted toward jungle." The result is a new, longer, sharper query that expresses what you meant rather than what you said.

THE STORY

Relevance feedback is one of the oldest ideas in information retrieval, worked out by J. J. Rocchio in the 1960s and 70s as part of the SMART retrieval system at Cornell. What US5987457 industrialises, three decades later, is applying it to document search in the commercial systems of the late 1990s — the enterprise search tools and early web search where users typed short, ambiguous queries and needed the system to help them home in. Its descendants are subtle but everywhere: the "related searches" and "did you mean" suggestions, the way modern search silently expands your query with synonyms and related terms, the "more like this" button, and the click-through models that treat every click as an implicit relevance judgement feeding exactly this kind of refinement. Every time a search engine seems to understand what you meant better than what you typed, some heir of Rocchio is at work.

THE CODE

The refinement is vector arithmetic: original query, plus a weighted centroid of the relevant documents, minus a weighted centroid of the irrelevant ones. Here is the core of `patentml`'s `query_refine`:

```
def refine(query, relevant, nonrelevant=(), alpha=1.0, beta=0.75,
gamma=0.25):
    q = _vec(query)
    rel = [_vec(d) for d in relevant]
    non = [_vec(d) for d in nonrelevant]
```

```
out = {}
for t in _all_terms(q, rel, non):
    w = alpha q[t] + beta _centroid(rel, t) - gamma *
    _centroid(non, t)
    if w > 1e-9:          # clamp negatives to zero
        out[t] = round(w, 6)
return out
```

The demo starts from the query *jaguar*, marks two car documents relevant and two animal documents irrelevant, and asks for the expansion terms. It prints:

Query refinement / Rocchio feedback (from US5987457, expired):

```
query: jaguar
relevant -> ['jaguar car engine luxury', 'jaguar luxury sedan car']
non-rel -> ['jaguar animal jungle cat', 'jaguar cat wildlife']
expansion -> ['car', 'luxury', 'engine', 'sedan']
```

The system read your reaction and inferred your intent. From two clicks it decided the query should grow the terms *car*, *luxury*, *engine*, *sedan* — the words that distinguish the car documents — while the animal words (*jungle*, *cat*, *wildlife*), heavy in the documents you rejected, were pushed to zero and never made the list. The next search with this expanded query would bury the animal results. The module verifies the key property directly — that relevant terms

outweigh irrelevant ones and negatives are clamped — and reports PASS.

THE INHERITANCE

Query refinement is public domain, and it is the tool for any search where the user's first words are only an approximation of their intent — which is almost every search. A few marks of "yes, more like this" and "no, not that," fed through this vector arithmetic, turn a vague query into a precise one without asking the user to be a better wordsmith. You can bolt it onto the inverted index of Chapter 4 in a handful of lines and get a search that improves as it is used, entirely locally, with no model to train and no service to call. The idea that a system should learn what you meant from how you react to what it showed you is one of the most humane in all of search, and Rocchio's sixty-year-old insight, industrialised in this 1997 filing, is now yours to build on freely.

CHAPTER 12

ONE INDEX OVER MANY FIELDS (INTEGRATED METADATA SEARCH)

THE FILING

US Patent 6,480,835 — "Method and system for searching on integrated metadata."

Filed 1998. Expired; public domain.

Real records are not just text; they carry structured metadata — a file has a type, a size, a date; a photo has a camera, a resolution, a location. Searching them well means combining conditions across fields of different kinds at once: "JPEGs, over one megabyte, taken in 2003." This patent's claims describe one integrated index over all the fields, keyed by field-and-value, so that such a multi-field query is resolved by intersecting per-field result sets rather than scanning every record. It is the structured-search engine behind every "advanced search" and filter sidebar you have ever used.

THE IDEA

Metadata search is the marriage of two ideas from earlier chapters. From the inverted index (Chapter 4) it borrows the trick of mapping values to the records that have them; from the composite index (Chapter 3) it borrows the discipline of typed, multi-column querying. The result handles two kinds of field differently, because they ask different questions.

Categorical fields — type, format, camera model — are asked with equality: "type equals JPEG." For these the index keeps, per field, a map from each value to the set of record ids that have it, exactly like a postings list. Ordered fields — size, date, price — are asked with ranges: "size between one and three megabytes." For these the index keeps the values sorted, so a range is a binary-search slice (Chapter 2's move).

A full query is a conjunction — several conditions that must all hold. The engine resolves each condition to a set of matching record ids, then intersects them: records that satisfy every condition survive. "JPEG and 2000–2004 and over 1MB" becomes three set-lookups and two intersections — no record ever scanned, no field ever read for a record that already failed an earlier condition. One integrated index, many field types, answered by set arithmetic.

THE STORY

By the late 1990s the world was filling with richly-tagged digital objects — documents, images, media files, web resources — each dragging a cloud of metadata behind it, and users increasingly wanted to search on that metadata rather than (or as well as) the content. US6480835 addresses the design question that raises: rather than a separate index per field with awkward glue between them, build one integrated index that understands field types and resolves cross-field conjunctions natively. Its descendants are the faceted-search and filtering systems that now feel mandatory: the sidebar of checkboxes and sliders on every shopping site, the "search by type, size, date modified" of a file manager, the advanced-search forms of digital libraries and photo tools. Every time you narrow a result set by ticking three boxes and dragging a slider, an integrated metadata index is intersecting sets behind the glass.

THE CODE

Categorical conditions become set lookups; ordered conditions become sorted-range slices; a query intersects them all. Here is the query resolver from `patentml`'s `metadata_search`:

```
def query(self, eq=None, ranges=None):
    result = None
    for f, v in (eq or {}).items():          # categorical: equality
        ids = set(self._eq[f].get(v, set()))
        result = ids if result is None else result & ids
```

```
for f, (lo, hi) in (ranges or {}).items():    # ordered: range slice
    ids = self._range_ids(f, lo, hi)
    result = ids if result is None else result & ids
    return sorted(result, key=repr) if result is not None else
sorted(self._ids, key=repr)
```

The demo indexes three files with type, year, and size, then asks two multi-field questions. It prints:

Integrated metadata search (from US6480835, expired):

```
type=jpeg AND year 2000..2004 -> ['f1']
size 100..300                -> ['f1', 'f3']
```

The first query combined a categorical equality (type = jpeg) with an ordered range (year 2000–2004) and intersected the two sets down to the single file that satisfied both — f2 is a JPEG but from 2005, so the year condition removed it. The second query, a pure range on size, returned the two files inside the band. Every answer came from set lookups and intersections, never a record scan. The module fuzz-tests three thousand record sets, checking equality-plus-range conjunctions against a full scan, and reports PASS.

THE INHERITANCE

Integrated metadata search is public domain, and it is the engine you

want the moment your records have structured fields people will filter on — which is almost any real dataset. A photo library, a document store, a product catalogue, a music collection: build one field-aware index and every "advanced search" and facet sidebar falls out as set intersection, fast and local. It composes naturally with the text search of Chapter 4 — content in the inverted index, attributes in the metadata index, a query touching both — to give the full "search and filter" experience that users now expect, all in dependency-free code small enough to run beside the data it indexes. The faceted search that big platforms sell as a feature is, underneath, a handful of sets being intersected, and that handful is now yours.

CHAPTER 13

MERGING THE SEARCH ENGINES (RANK FUSION)

THE FILING

US Patent 6,795,820 — "Metasearch technique that ranks documents obtained from multiple collections."

Filed 2001. Expired; public domain.

Sometimes one search engine is not enough — you want the combined judgement of several, each searching its own collection, each returning its own ranked list. But their lists disagree, and their scores are not comparable: one returns relevance from 0 to 1, another a mysterious integer, a third only an order with no scores at all. This patent's claims describe fusing those separate ranked lists into a single, better ranking. It is the algorithm behind metasearch — the search engine of search engines — and behind every modern system that blends results from multiple retrievers.

THE IDEA

You asked the same question of three search engines and got three ranked lists back. How do you merge them into one list that is better than any single engine's? You cannot simply compare scores — engine A's "0.9" and engine B's "0.9" mean nothing to each other. The trick is to fuse on rank (position) rather than raw score, so that all engines speak a common language: "you put this first, they put it second."

Three classic fusion methods, all in `patentml`'s module, capture the idea from different angles:

- Reciprocal Rank Fusion (RRF) gives each document a score of $1/(k + \text{rank})$ from each list — high for things near the top, gently diminishing further down — and sums across lists. It is robust, needs no scores at all, and is the modern default.
- Borda count, borrowed from voting theory, treats each list as an election: in a list of n items, position i awards $n - i + 1$ points across all lists; the document with the most "votes" wins.
- CombSUM is for when the engines do give comparable-once-normalised scores: min-max normalise each list to a common 0–1 scale, then add the scores.

All three share the same wisdom: a document that several engines rank highly should beat a document that only one engine loved.

Consensus near the top is the signal; disagreement is noise to be averaged out.

THE STORY

Metasearch was a real and visible product in the late 1990s and early 2000s — engines like MetaCrawler and Dogpile that had no index of their own but queried several others and blended the results, on the reasonable theory that no single search engine covered the whole web. US6795820, filed in 2001, is a patent from that world, concerned precisely with the fusion step: how to rank documents pulled from multiple collections whose scores you cannot trust to be comparable. The idea outlived the metasearch products. Rank fusion — and RRF in particular — became a cornerstone of modern information retrieval, the standard way to combine a keyword search with a semantic (vector) search, to blend multiple rankers in a recommendation system, or to pool the outputs of an ensemble of retrievers in the retrieval-augmented systems behind today's AI. The metasearch engines faded; their fusion mathematics is more central than ever.

THE CODE

Each method turns positions into scores and sums across lists. Here are RRF and Borda from `patentml`'s `rank_fusion`:

```
def reciprocal_rank_fusion(rankings, k=60):
    score = defaultdict(float)
    for lst in rankings:
        for rank, doc in enumerate(lst):
            score[doc] += 1.0 / (k + rank + 1)    # top ranks worth most
    return _sorted(score)

def borda(rankings):
    score = defaultdict(float)
    for lst in rankings:
        n = len(lst)
        for i, doc in enumerate(lst):
            score[doc] += n - 1 - i              # position i earns n-1-i votes
    return _sorted(score)
```

The demo fuses three disagreeing engine results and also a scored pair. It prints:

Rank fusion (from US6795820, expired):

RRF -> ['b', 'a', 'c', 'e', 'd']

Borda -> ['b', 'a', 'c', 'd', 'e']

CombSUM (normalized) -> ['b', 'a', 'c']

Look at what won. Document b was never ranked first by any single engine, yet it comes first in the fused ranking — because it appeared

near the top of all three, and consensus beats any single engine's top pick. Document a, a first-place in one list but lower in others, lands second. The two rank-based methods (RRF, Borda) broadly agree while breaking their lower-order ties slightly differently, exactly as the theory predicts. The module verifies the defining properties — a document ranked first everywhere must win, the Borda arithmetic matches by hand, and fusing a single list returns it unchanged — and reports PASS.

THE INHERITANCE

Rank fusion is public domain, and it is one of the most useful and under-appreciated tools a builder can own. The instant you have more than one way to rank things — keyword search and semantic search, two recommendation signals, several models voting — RRF blends them into something better than any alone, with no training, no score calibration, and about ten lines of code. It is the quiet workhorse inside modern "hybrid search" and retrieval pipelines, and it works just as well fusing two tiny local rankers on a Raspberry Pi as it does pooling a fleet of engines. Whenever you catch yourself with several opinions and no principled way to combine them, this chapter is the answer: trust the consensus near the top, and let disagreement average away.

CHAPTER 14

SPLITTING THE SEARCH (PARALLEL PARTITIONED SEARCH)

THE FILING

US Patent 5,884,303 — "Parallel searching technique."

Filed 1997. Expired; public domain.

Eventually one machine is not enough — the data outgrows a single computer, or a single computer's search is too slow, and you must spread the work across many. This patent's claims describe partitioning the data across independent shards, keyed by a hash of each item, so that a search runs on every shard at once and the partial results are merged. The answer is identical to a single scan; the win is that each shard touches only its own slice. It is the map-then-merge pattern that underlies every distributed search and database in the world.

THE IDEA

You have a billion items and one search is too slow, or the items no longer fit on one machine. The fix is to divide and conquer: split the data into pieces — shards — spread them across many workers, and let each worker search only its own piece. Because the workers run at the same time, the search finishes in the time it takes one worker to scan one shard, not one machine to scan everything. Then you merge the workers' partial answers into the final result.

The critical decision is how to split the data, and the answer is to split by a hash of the item. Hashing scatters items evenly across shards, so each shard gets roughly the same share of the data and the same share of the work — no single shard becomes a bottleneck. It also makes point lookups cheap: to find one specific item, hash it and you know exactly which shard holds it, so you ask that one shard and ignore all the others. A broad search fans out to every shard in parallel; a targeted lookup goes to exactly one. Either way, the merged result is provably identical to what a single machine scanning everything would have returned — the parallelism changes the speed, never the answer.

THE STORY

Parallel and distributed search is foundational to large-scale computing, and by the late 1990s it was moving from

supercomputers into mainstream databases and search systems that had outgrown single machines. US5884303 is a filing from that transition, capturing the essential pattern: hash-partition the data, search the shards independently, merge the results. That pattern is one of the most consequential in all of computing. A few years later it was generalised and made famous as MapReduce — "map" a function across shards, "reduce" (merge) the results — which powered the indexing of the entire web and launched the era of big data. Every distributed database that "scales horizontally," every sharded search cluster, every "add more machines to go faster" system, is running some refinement of the split-search-merge idea this 1997 patent describes. It is the reason data can grow without a query grinding to a halt: you grow the machines alongside it.

THE CODE

The pattern is: hash each item to a shard on the way in; on a broad search, scan every shard and merge; on a lookup, go to the one shard that can hold it. Here is the core of `patentml`'s `parallel_search`:

```
def _shard_of(self, item):  
    return (hash(item) & 0x7FFFFFFF) % self.n    # even scatter  
across shards  
  
def search(self, predicate):  
    merged = []
```

```
for sh in self.shards:          # each shard, independently
    merged.extend(x for x in sh if predicate(x))
return sorted(merged, key=repr)  # merge the partial
results
```

```
def find(self, item):
    return item in self.shards[self._shard_of(item)] # lookup hits
ONE shard
```

The demo spreads the numbers 0–19 across four shards, then runs a broad search and two lookups. It prints:

Parallel partitioned search (from US5884303, expired):

```
evens in 0..19 -> [0, 10, 12, 14, 16, 18, 2, 4, 6, 8]
per-shard hit counts: [5, 0, 5, 0] (work split across shards)
find(13) -> True  find(99) -> False
```

The broad "find all evens" search ran across all four shards and merged their hits into the complete, correct set — identical to what a single scan would give. The per-shard counts [5, 0, 5, 0] show the work genuinely divided: two shards did the finding, in parallel, each touching only its own fifth of the data. And the two lookups went to a single shard each — find(13) found it, find(99) correctly reported its absence — without disturbing the others. The module fuzz-tests four thousand datasets across varying shard counts, checking the

merged search and single-shard lookups against a plain linear scan, and reports PASS.

THE INHERITANCE

Parallel partitioned search is public domain, and it is the mental model behind scaling anything past one machine. The lesson is more valuable than the code: split by a hash for even load, let the pieces work independently, and merge — and your answer is unchanged while your capacity grows with the hardware you add. You meet this pattern the moment a dataset outgrows a laptop, and understanding it demystifies the whole vocabulary of "sharding," "horizontal scaling," and "MapReduce" that surrounds big systems. Even at small scale it teaches the right instinct: to go faster or bigger, divide the work so the pieces don't wait on each other. That instinct, and a clean split-search-merge to build on, is the inheritance of this 1997 filing — the seed of the idea that let data grow to the size of the web.

CHAPTER 15

SAME PERSON, DIFFERENT SPELLING (RECORD MATCHING AND LINKAGE)

THE FILING

US Patent 6,026,398 — "System and methods for searching and matching databases."

Filed 1997. Expired; public domain.

Every exact-match technique in this book assumes the data is clean. Real data is not. The same person is "Robert Smith" in one system and "Bob Smith" in another; the same company is "Acme Ltd" and "ACME Limited"; the same address is spelled three ways. This patent's claims describe matching records that refer to the same entity despite such differences, by scoring the similarity of their fields and linking pairs whose score crosses a threshold. It is the primitive behind deduplication, entity resolution, and the merging of databases that were never designed to agree.

THE IDEA

Exact matching fails the moment data comes from more than one source, because humans and systems spell the same thing differently. Record matching replaces the yes/no of equality with a degree of similarity, a number between 0 (nothing alike) and 1 (identical), and links records whose similarity is high enough.

Two complementary measures do the scoring. Token overlap (the Jaccard measure) breaks each record into words and asks what fraction of the combined word-set they share: "Robert Smith" and "Bob Smith" share smith out of {robert, smith, bob}, a partial overlap. Edit distance counts the single-character insertions, deletions, and substitutions needed to turn one string into the other, then turns that into a similarity ratio: "Acme Ltd" and "ACME Limited" are a few edits apart out of a dozen characters, so fairly similar. Blending the two — one catches reordered and abbreviated words, the other catches typos and spelling drift — gives a robust score. Any pair scoring above a chosen threshold is declared a match; the threshold is the dial that trades false matches against missed ones.

That single idea — score similarity, link above a threshold — is the whole of what the industry calls entity resolution: deciding which records, scattered across systems and riddled with variation, actually describe the same real-world thing.

THE STORY

Record linkage is older than computing as a formal discipline — statisticians Newcombe, and later Fellegi and Sunter, laid its probabilistic foundations in the 1950s and 60s for linking medical and census records. US6026398 is the late-1990s industrialisation of that lineage inside database systems, as organisations increasingly needed to merge customer lists, deduplicate mailing databases, and reconcile records across the many systems a large company accumulates. The problem only grew. Today entity resolution is a field and an industry of its own: the "we found your account" that recognises you across a company's siloed systems, the deduplication that stops you being mailed twice, the fraud detection that spots one person behind five slightly-different identities, the master-data management that gives a business a single view of each customer. Every one of them rests on the same move this patent describes — score the similarity, link above the line — and its moat, then and now, is getting that scoring right.

THE CODE

Two similarity measures, blended, above a threshold. Here is the scoring core of `patentml`'s `record_match`:

```
def jaccard(a, b):                # shared words / all words
    ta, tb = tokens(a), tokens(b)
```

```
    return len(ta & tb) / len(ta | tb) if (ta or tb) else 1.0

def similarity(a, b):          # blend token overlap with edit
similarity
    return 0.5 jaccard(a, b) + 0.5 edit_ratio(a, b)

def match(left, right, threshold=0.6):
    out = [(i, j, round(similarity(a, b), 4))
            for i, a in enumerate(left)
            for j, b in enumerate(right) if similarity(a, b) >= threshold]
    out.sort(key=lambda t: -t[2])    # best matches first
    return out
```

The demo matches a list of names and companies against a second, dirtier list. It prints:

Record matching / linkage (from US6026398, expired):

```
'Robert Smith' ~ 'Bob Smith'  score=0.5
'Acme Ltd' ~ 'ACME Limited'  score=0.5
'Jon A Doe' ~ 'John Doe'  score=0.4583
```

Three genuine matches, none of them exact. "Robert Smith" linked to "Bob Smith" on their shared surname and overlapping characters; "Acme Ltd" to "ACME Limited" across a case change and an abbreviation; "Jon A Doe" to "John Doe" despite a dropped middle

initial and a spelling variation. Each score reflects how strong the evidence is, and sorting by score puts the most confident links first — which is exactly how a human reviewer would want them queued. The module verifies the measure's properties — identical strings score 1.0, disjoint strings fall below the threshold, and the score is symmetric — and reports PASS.

THE INHERITANCE

Record matching is public domain, and it is the tool for the messy reality that exact matching ignores: merging two contact lists, deduplicating a customer database, reconciling records that came from different systems and never agreed on spelling. A blend of token overlap and edit distance, a threshold you tune to your tolerance for false matches, and you have real entity resolution in a single file — no service, no model, running privately over data that often should never leave your machine. It pairs naturally with the search of Chapter 4 and the metadata index of Chapter 12 to turn a pile of inconsistent records into a clean, linked whole. The unglamorous work of deciding that two spellings are one person is among the most valuable in all of data — and now that this filing has expired, its foundation is yours. With that, Part III is complete: you can join tables, keep answers ready, question a moving stream, refine a vague search, filter across fields, merge many engines, split work across machines, and match what should be matched. The database has learned to answer.

PART IV — ORDER AND FILTERS

The queries of Part III answered questions about sets of rows: join them, count them, group them. This part turns to two subtler jobs. The first is order statistics — not "how many" but "what is the middle value," "the ninetieth percentile," "the k-th largest" — questions about a value's rank that a naïve system answers only by sorting everything, again, every time. The second is filtering at scale — routing a flood of incoming documents to the many standing queries that want them, and matching records that are alike in sound or in shape rather than in exact spelling. Four chapters: the ranked tree, the shared-predicate filter, phonetic matching, and document fingerprinting. Each replaces a brute-force scan with a structure that already knows the answer's shape.

CHAPTER 16

THE K-TH SMALLEST (ORDER-STATISTIC TREES)

THE FILING

US Patent 5,875,445 — "Performance-related estimation using pseudo-ranked trees."

Filed 1996. Expired; public domain.

Some of the most-asked questions in analytics are about rank: the median response time, the ninety-fifth-percentile latency, the top-k earners, "how many customers spend less than this one." Answering them by sorting the whole dataset every time is wasteful when the data keeps changing. This patent's claims describe a search tree whose nodes carry the size of their subtrees, so it can answer "what is the k-th smallest?" and "how many fall below x?" directly, without a scan or a re-sort. It is the structure that makes percentiles cheap on data that will not hold still.

THE IDEA

A plain search tree (Chapter 1) can find whether a value is present, but it cannot easily tell you a value's rank — its position in sorted order — because it does not know how many values sit to its left. An order-statistic tree fixes that with one small addition: every node records the size of its subtree, the count of values at or below it. That single number, maintained as the tree changes, unlocks two operations that otherwise demand a full sort.

Select(k) — "what is the k -th smallest value?" — walks down from the root, using the subtree sizes as a compass. At each node, the size of its left subtree tells you how many values are smaller. If k is less than that, the answer is in the left subtree; if k equals it, this node is the k -th; otherwise go right, subtracting the values you have skipped. A handful of steps down the tree lands exactly on the k -th smallest, with no sorting.

Rank(x) — "how many values are below x ?" — is the mirror image: descend toward x , and every time you turn right, add the left subtree's size plus one, accumulating the count of everything smaller. From select and rank, everything else follows: the median is select($n/2$), any percentile is select, and a value's standing in the population is rank. All in the depth of the tree rather than the size of the data, and all correct even as values are continuously inserted.

THE STORY

The order-statistic tree is a classic of the algorithms literature — an augmented balanced tree, the textbook example of how adding a little bookkeeping to each node buys a whole new class of query. US5875445's framing, "performance-related estimation," points at why industry cared in the late 1990s: monitoring systems needed to track the distribution of measurements — response times, transaction latencies, resource usage — and report medians and percentiles continuously, over streams of numbers that never stopped arriving. Sorting the whole history for every dashboard refresh was untenable; a ranked tree gave the percentiles directly and stayed correct as each new measurement landed. Its descendants run in every performance-monitoring and observability tool that shows you a "p50 / p95 / p99" latency, in the leaderboards that report your rank among millions, and in any system that must answer "where does this value stand among all the others" without stopping to sort them.

THE CODE

The magic is `select`, steering by subtree sizes. Here it is from `patentml`'s `rank_tree` (a size-augmented randomized tree):

```
def select(self, k):          # the k-th smallest, 0-indexed
    n = self.root
    while n:
```

```
left = _sz(n.l)      # how many values are smaller
if k < left: n = n.l  # answer is to the left
elif k == left: return n.key # this node is the k-th
else:               k -= left + 1; n = n.r # skip left+self, go right
```

The demo inserts eight numbers in scrambled order, then asks rank questions. It prints:

Order-statistic (pseudo-ranked) tree (from US5875445, expired):

```
select(0),(3),(7) -> 10 40 90
rank(35)          -> 3
median            -> 40
```

The numbers went in as 50, 20, 70, 10, 30, 60, 90, 40. Without ever sorting them into a list, the tree reports that the smallest (`select(0)`) is 10, the fourth-smallest (`select(3)`) is 40, and the largest (`select(7)`) is 90 — reading them straight off the subtree sizes. `rank(35)` correctly answers that three stored values (10, 20, 30) fall below 35. And the median is `select(3) = 40`. The module fuzz-tests four thousand datasets, checking every select and rank against a freshly sorted list, and reports PASS.

THE INHERITANCE

The order-statistic tree is public domain, and it is the right structure

whenever you need ranks, percentiles, or medians over data that keeps changing. Reach for it in monitoring and observability (live p95 latency), in leaderboards and rankings, in any streaming statistic where re-sorting the history for each query would be absurd. It is the clearest lesson in the whole book about augmentation — that a modest annotation on each node of a familiar structure (here, a subtree count) can unlock an entire new class of question at almost no cost. Once you have seen it, you start noticing where else a little bookkeeping would turn a scan into a walk. That instinct, and a ready-made ranked tree, are yours now, a quarter-century after the filing that put percentiles within a single traversal's reach.

CHAPTER 17

ONE TEST, MANY SUBSCRIBERS (THE DAG DOCUMENT FILTER)

THE FILING

US Patent 5,873,081 — "Document filtering via directed acyclic graphs."

Filed 1997. Expired; public domain.

Every query so far has been asked of stored data. This chapter inverts the relationship: the queries are stored, standing and waiting, and the documents flow past to be routed to whichever queries want them. When thousands of standing queries share conditions — "UK news," "long UK news," "news about sport" all test "is this news?" — testing every condition for every query is enormous waste. This patent's claims describe compiling the queries into a directed acyclic graph of shared predicates, so each condition is evaluated once per document no matter how many queries depend on it. It is the engine

of publish-subscribe.

THE IDEA

Picture a newswire and ten thousand people who have each registered a standing interest: one wants UK news, one wants long UK news, one wants sport, one wants long sport from Europe. A new article arrives. Which subscribers should get it? The naïve answer evaluates every subscriber's conditions from scratch — and immediately notices that thousands of them begin by asking the same thing: "is this news?", "is this about the UK?" Re-testing those shared conditions once per subscriber is the waste, and it grows with the audience.

The DAG filter eliminates it by recognising that the subscribers' conditions form a graph with enormous overlap. Compile them into a directed acyclic graph whose nodes are predicates ("is news," "is UK," "over 500 words") and Boolean combinations (AND, OR) of other nodes — and, crucially, merge shared sub-expressions into a single node. "UK news" and "long UK news" both point at the same "is UK" node and the same "is news" node. When a document arrives, evaluate the graph once, bottom-up, memoising each node's result: every shared predicate fires exactly one time, its answer reused by every subscriber that depends on it. Ten thousand subscribers sharing three underlying conditions cost three predicate evaluations, not thirty thousand.

THE STORY

This is the core algorithm of publish-subscribe systems and content-based routing — a field that grew urgent in the late 1990s as the volume of streaming information (news feeds, market data, alerts, later email and notifications) outstripped any ability to match it against subscribers one query at a time. US5873081 captures the essential move: don't run the subscriptions; compile them into a shared evaluation network and run the document through it once. Its intellectual cousins are the Rete algorithm that powers rule engines and the predicate-matching networks inside message brokers. Its descendants are everywhere information is routed by interest rather than address: the notification systems that decide which of a billion users cares about an event, the alerting rules in monitoring platforms, the topic routers in message queues, the ad and content targeting that matches items to audiences. Whenever a system efficiently answers "who wants this?" over many overlapping standing queries, a shared-predicate graph is doing the work.

THE CODE

The evaluation is one memoised pass over the DAG, so each shared predicate fires once. Here is the matcher from `patentml`'s `dag_filter`:

```
def match(self, doc):  
    memo = {}
```

```
self._evals = 0
def ev(nid):
    if nid in memo:          # shared node already computed —
reuse
        return memo[nid]
    op, arg = self._nodes[nid]
    if op == "pred":
        self._evals += 1    # a real predicate test happens here
        res = bool(arg(doc))
    elif op == "and": res = all(ev(c) for c in arg)
    else:      res = any(ev(c) for c in arg)
    memo[nid] = res
    return res
return [lbl for lbl, root in self._subs.items() if ev(root)]
```

The demo registers two subscriptions — "UK news" and "long UK news" — that share the `is_news` and `is_uk` predicates, then routes one document. It prints:

DAG document filter (from US5873081, expired):

```
doc: {'topic': 'news', 'region': 'uk', 'words': 800}
matches -> ['A: uk news', 'B: long uk news']
predicate evaluations (3 shared preds, not 5): 3
```

The document matched both subscriptions — it is UK news, and it is

long. But the number that matters is the last one: 3 predicate evaluations, not 5. The two subscriptions between them reference five predicate-slots (`is_news`, `is_uk` for A; `is_news`, `is_uk`, `is_long` for B), but the shared `is_news` and `is_uk` were each tested only once and their results reused. With three subscriptions the saving is larger; with ten thousand it is the difference between feasible and hopeless. The module fuzz-tests five thousand documents, checking the DAG's matches against evaluating each subscription independently, and reports PASS — sharing changes the cost, never the answer.

THE INHERITANCE

The DAG document filter is public domain, and it is the tool for the "who wants this?" problem — routing a stream of items to many standing, overlapping queries. Notification systems, alerting rules, content and topic routing, feed personalisation: all of them face the same explosion of redundant predicate tests, and all of them are tamed by compiling the queries into a shared graph evaluated once per item. It is also a small masterclass in a deep idea — that common sub-expression elimination, the compiler's trick of computing a shared thing once, applies far beyond compilers. Anywhere many computations overlap, a DAG that merges the shared parts turns quadratic waste into linear work. That pattern, and a working publish-subscribe filter to build on, are yours now that this 1997 filing has expired.

CHAPTER 18

SOUNDS LIKE (PHONETIC SEARCH)

THE FILING

US Patent 6,408,270 — "Phonetic sorting and searching."

Filed 6 October 1998. Expired 6 October 2018; public domain.

People misspell names constantly, and a search that only matches exact spellings fails them constantly. "Was it Robert or Rupert? Smith or Smyth? Ashcraft or Ashcroft?" This patent's claims describe converting each name to a phonetic code — a term drawn from a small finite set, capturing how the name sounds rather than how it is spelled — and matching on that code, so that names which sound alike are retrieved together however they are written. It is the reason a search for "Robbert" can still find "Robert."

THE IDEA

The insight is to stop matching letters and start matching sound. A phonetic algorithm reduces a name to a short code such that names pronounced alike collapse to the same code, while names that sound different get different codes. Index by the code, and a search reduces its own input to a code and looks up the bucket — retrieving every name that sounds like it, regardless of spelling.

The classic scheme, over a century old, is Soundex: keep the first letter, then encode the remaining consonants into digits by articulation group (b, f, p, v all become 1; c, g, j, k, q, s, x, z become 2; and so on), drop the vowels, collapse adjacent duplicates, and pad or trim to a fixed length. "Robert" and "Rupert" both become R163; "Smith" and "Smyth" both become S530; "Ashcraft" and "Ashcroft" both become A261. The spellings differ; the sounds — and therefore the codes — are the same. To make the search rank its results, patentml's module pairs the phonetic bucket with an edit-distance tie-break, so that among the names sharing a code, the ones closest in actual spelling to the query surface first.

THE STORY

Soundex is one of the oldest algorithms in this entire series — patented by Margaret Odell and Robert Russell in 1918 and 1922, and used for decades to index the US census so that clerks could find a family whatever the enumerator's spelling of the surname. It is genuinely public-domain heritage. What US6408270 industrialises,

eighty years later, is phonetic sorting and searching inside the digital systems of the late 1990s — applying the sound-alike idea to the databases, directories, and search boxes where users typed names they could not spell. Its descendants are the "did you mean?" behind every name search, the fuzzy matching in contact lookups and customer-service systems, the genealogy sites that find your ancestors across centuries of creative spelling, and the more modern phonetic schemes (Metaphone, NYSIIS, Double Metaphone) that refine Soundex's rough edges. Every time a system finds the person you meant despite the letters you typed, a phonetic code is doing the work Odell and Russell invented a hundred years ago.

THE CODE

The whole trick is the Soundex encoder; search is a lookup of the query's code plus an edit-distance ranking. Here is the encoder's heart from `patentml`'s `phonetic_search`:

```
def soundex(name):
    name = name.upper()
    first = name[0]
    codes = {dict.fromkeys("BFPV", "1"),
dict.fromkeys("CGJKQXZ", "2"),
            dict.fromkeys("DT", "3"), "L": "4",
            dict.fromkeys("MN", "5"), "R": "6"}
    tail = []
```

```
prev = codes.get(first, "")
for ch in name[1:]:
    c = codes.get(ch, "")
    if c and c != prev:      # skip vowels; collapse adjacent
duplicates
        tail.append(c)
    prev = c
return (first + "".join(tail) + "000")[:4]
```

The demo encodes several sound-alike pairs, then searches for two misspellings. It prints:

Phonetic search (from US6408270, expired):

```
soundex(Robert)=R163 soundex(Rupert)=R163 match=True
soundex(Ashcraft)=A261 soundex(Ashcroft)=A261 match=True
soundex(Smith)=S530 soundex(Smyth)=S530 match=True
soundex(Tymczak)=T522 soundex(Tymczak)=T522 match=True

search('Robbert') -> ['Robert', 'Rupert']
search('Smitt') -> ['Smith', 'Smythe']
```

The pairs show the collapse: Robert and Rupert, spelled differently, both reduce to R163 and are declared a match; the same for the other pairs. Then the searches deliver the payoff — a query for the misspelled "Robbert" retrieves the correctly-spelled "Robert" and the

sound-alike "Rupert," and "Smitt" finds both "Smith" and "Smythe." None of those were exact matches; all were found by sound. The module's checks confirm the encodings and matches, reporting PASS.

THE INHERITANCE

Phonetic search is public domain — indeed its Soundex core has been for a century — and it remains the simplest way to make any name search forgiving. Bolt it onto a contact list, a customer database, a directory, or a genealogy tool and suddenly misspelled queries find their targets, with no fuzzy-matching library and no model, in a few dozen lines that run anywhere. It pairs naturally with the record matching of Chapter 15 (sound-alike as one similarity signal among several) and with the inverted index of Chapter 4 (index the phonetic code as just another term). More than any other chapter, this one shows how long the tail of a good idea is: an algorithm built to index a paper census in 1918 is still the right answer for a search box in 2026 — and the 1998 filing that carried it into the digital age has now expired into your hands.

CHAPTER 19

THE SHAPE OF A DOCUMENT (DOCUMENT FINGERPRINTING)

THE FILING

US Patent 6,363,381 — "Compressed document matching."
Filed 1998. Expired; public domain.

Chapter 18 matched names that sound alike. This one matches documents that look alike — the same document scanned twice, a form filled in two ways, a page and its near-duplicate — by reducing each to a compact numeric fingerprint and comparing the fingerprints instead of the documents. The claims describe matching documents in a compressed, summarised form rather than by comparing them in full, so that "is this the same document?" becomes a fast comparison of small signatures. It is the primitive behind duplicate detection at scale.

THE IDEA

Comparing two documents character by character is slow, and useless the moment the differences are cosmetic — a re-scan with slightly different noise, a form with the same structure but different ink, two copies with trivial edits. What you usually want is not "are these byte-identical?" but "are these the same document?" — and that question is best answered by comparing summaries.

A fingerprint is a small fixed-size numeric summary of a document's structure — in `patentml`'s module, a profile of the document's layout and content statistics — designed so that similar documents produce similar fingerprints and different documents produce different ones. Reduce every document to its fingerprint once; then "does this match anything I've seen?" becomes a comparison of small vectors rather than a comparison of full documents. The comparison itself uses normalised cross-correlation — a similarity score, robust to overall shifts in scale, that returns near 1.0 for documents of the same shape and lower values as they diverge. A document compared to itself scores 1.0; a near-duplicate scores high; an unrelated document scores low. Store the fingerprints in an index and you can ask "what have I seen that looks like this?" over a huge collection cheaply, because you never touch the documents themselves — only their signatures.

THE STORY

Document fingerprinting grew out of a very practical late-1990s need: organisations were digitising and storing documents in enormous volumes, and drowning in duplicates — the same page scanned repeatedly, the same form submitted twice, near-identical versions piling up. US6363381's "compressed document matching" addresses exactly that, matching documents in a summarised form so the duplicates can be found and collapsed without exhaustively comparing everything to everything. The idea generalised far beyond scanned paper. Its descendants are the near-duplicate detection that keeps a search engine from showing you the same page ten times, the similarity hashing (MinHash, SimHash) that finds near-duplicates across billions of web pages, the plagiarism detectors that compare documents by fingerprint, and the perceptual hashes that match images and audio despite re-encoding. The unifying move — summarise, then compare the summaries — is one of the most scalable ideas in all of information management, and this filing is one of its database-era expressions.

THE CODE

Reduce each document to a profile, then score two profiles by normalised cross-correlation. Here is the comparison core of patentml's `document_fingerprint`:

```
def ncc(a, b):                # normalised cross-correlation
    n = min(len(a), len(b))
```

```
a, b = a[:n], b[:n]
ma, mb = sum(a)/n, sum(b)/n
num = sum((a[i]-ma)*(b[i]-mb) for i in range(n))
da = (sum((x-ma)2 for x in a))0.5
db = (sum((x-mb)2 for x in b))0.5
return num/(da*db) if da and db else 0.0 # ~1.0 = same shape
```

The demo fingerprints three documents and ranks them against document A. It prints:

Document fingerprint (from US6363381, expired):

```
query(A) ranked -> [('A', 1.0), ('B', 0.939), ('C', 0.775)]
best match for A is A at 1.000 (self, ~1.0)
```

Read the ranking. Document A compared to itself scores exactly 1.0 — a fingerprint always matches its own document perfectly, the property that makes the technique trustworthy. Document B, a near-relative of A, scores 0.939 — high, flagged as very similar. Document C, more distant, scores 0.775. The whole comparison happened on small numeric signatures, never on the documents themselves, so the same ranking would run just as fast over a million stored fingerprints. The module fuzz-tests the matcher and reports PASS, confirming a document's self-match tops every ranking.

THE INHERITANCE

Document fingerprinting is public domain, and it is the tool for the deduplication and near-duplicate problems that quietly plague any large collection: scanned archives, form submissions, document stores, content pipelines, anywhere the same thing arrives more than once in slightly different clothes. Summarise each item to a compact fingerprint, index the fingerprints, and "have I seen this before, or something like it?" becomes cheap and scalable, running over signatures instead of originals. It is the database chapter of a family of ideas — perceptual hashing, similarity hashing, content-addressing — that recur throughout computing wherever "the same thing, roughly" matters more than "the same bytes exactly." With that, Part IV closes: you can rank a value among millions, route a stream to the queries that want it, match what sounds alike, and match what looks alike — the filters and order statistics that sit beyond the plain query. Part V turns to data that carries coordinates: space.

PART V — SPACE: DATA WITH COORDINATES

So far, the data in this book has had keys but no geometry — names, numbers, categories, arranged on a line. But an enormous amount of the world's data lives in space: points on a map, pixels in an image, coordinates in a simulation, anything with a position in two or more dimensions. Space breaks the one-dimensional tricks of Part I, because there is no single "sorted order" for points on a plane — sort by longitude and you scatter neighbours that share a latitude. This part is about the structures that tame multi-dimensional data: dividing space into cells, pruning the regions a query cannot touch, folding many dimensions onto a single line without losing locality, and turning positions on the Earth into keys you can index. Five chapters on the geometry of stored data.

CHAPTER 20

BOXES ON A GRID (INTERVAL PARTITIONING)

THE FILING

US Patent 6,003,036 — "Interval-partitioning method for multidimensional data."

Filed 12 February 1998. Expired 12 February 2018; public domain.

The first problem of spatial data is the range query: "find every point inside this rectangle." On a line you would binary-search to each end and read the slice between — but a rectangle has no single sorted order to search. This patent's claims describe splitting each dimension into intervals, forming a grid of cells, and indexing every point by the cell it falls in, so that a window query visits only the cells the window overlaps rather than scanning all the points. It is the grid index, the simplest structure that makes "what is near here" fast.

THE IDEA

You have a million points scattered on a plane and you want the ones inside a small rectangle. Checking all million against the rectangle is a full scan — fine once, ruinous as a habit. The grid index avoids it by imposing structure on the empty space itself.

Divide each axis into fixed-width intervals, so the plane is carved into a grid of rectangular cells. Every point belongs to exactly one cell, decided by simple division: floor the coordinates by the cell size. Store, for each cell, the list of points that fell in it. Now a window query is answered in two moves. First, compute which cells the query rectangle overlaps — a small block of the grid. Second, look inside only those cells, and among their points, keep the ones actually inside the rectangle. Every cell the window does not touch is skipped entirely, along with all its points. For a query that covers a small area, that is a handful of cells out of thousands — the difference between reading a million points and reading a few dozen.

The one design choice is the cell size, and it is a genuine trade-off. Cells too large and each holds many points, so you still check too many. Cells too small and a query spans many cells, adding overhead. Sized to the typical query, the grid turns "what is near here" from a scan into a lookup.

THE STORY

Spatial indexing became urgent in the 1990s as databases took on data with real geometry — geographic information systems mapping the physical world, CAD systems, scientific data, the beginnings of location-aware computing. The plain B-tree of Chapter 1 is helpless here, because it orders on one key and space has no single order. US6003036 is a filing from that spatial turn, capturing the most direct answer: quantise space into a grid and index points by cell. The grid is the simplest member of a whole family of spatial structures — R-trees, quadtrees, k-d trees, and the space-filling curves of Chapter 22 — that databases grew to handle coordinates. Its descendants run under every "find restaurants near me," every map that loads only the tiles you are looking at, every game that checks which objects are close enough to collide, and every spatial database that answers "what falls inside this boundary." Whenever a system answers a question about a region without examining the whole world, a spatial partition is why.

THE CODE

Bucket points by cell on the way in; on a window query, visit only the overlapping cells. Here is the range query from `patentml`'s `interval_partition`:

```
def range_query(self, lo, hi):
    lo_c = [int(lo[k] // self.cell[k]) for k in range(self.dims)]
    hi_c = [int(hi[k] // self.cell[k]) for k in range(self.dims)]
```

```
out = []
for cell in self._cells_in(lo_c, hi_c):    # only the overlapping cells
    for pt, payload in self.buckets.get(cell, []):
        if all(lo[k] <= pt[k] <= hi[k] for k in range(self.dims)):
            out.append(payload if payload is not None else pt)
return out
```

The demo places six points on a plane partitioned into 3×3 cells, then asks for everything inside a window. It prints:

Interval partitioning (from US6003036, expired 2018):

```
points      : [(1, 1), (2, 5), (8, 8), (4, 4), (9, 1), (5, 6)]
window (2,2)-(6,6): [(2, 5), (4, 4), (5, 6)]
```

The window from (2,2) to (6,6) correctly returned the three points inside it — (2,5), (4,4), (5,6) — and it found them by looking only in the cells the window overlaps. The points at (1,1), (8,8), and (9,1) live in cells the window never touches, so they were skipped without ever being tested against the rectangle. On six points the saving is invisible; on a million it is the whole ballgame. The module fuzz-tests three thousand point sets with random cell sizes and windows against a full scan, and reports PASS.

THE INHERITANCE

Interval partitioning is public domain, and it is the first tool to reach for the moment your data has coordinates and your questions have regions: points near a location, objects within a boundary, collision candidates in a game, samples inside a range across several axes. A grid, a bucket per cell, and a query that visits only the overlapping cells give you real spatial search in a few dozen lines that run on anything. It is also the gateway to spatial thinking — once you see space carved into cells you can visit selectively, the fancier structures (quadtrees, R-trees) become refinements of the same instinct rather than mysteries. That instinct, and a working grid index, are yours now that this 1998 filing has expired.

CHAPTER 21

SKIPPING WHAT CAN'T MATCH (CLUSTERED PARTITIONS AND ZONE MAPS)

THE FILING

US Patent 5,710,915 — "Method for accelerating access to a database by clustered partitioning."

Filed 1995. Expired; public domain.

Chapter 20 divided space into cells. This chapter divides a table into partitions and records, for each one, the range of values it contains — a tiny summary that lets a query skip whole partitions without reading a single row inside them. The claims describe clustering rows by a key into partitions and keeping each partition's minimum and maximum, so a range query consults those summaries and scans only the partitions whose range overlaps the query. It is the zone map, the technique that lets a modern analytic database read a hundred-billion-row table by touching a fraction of it.

THE IDEA

Suppose a huge table is stored in chunks — partitions of a few thousand rows each — and the rows are clustered by some key, so each partition holds a contiguous band of key values. Now, for each partition, remember just two numbers: the smallest and largest key it contains. That pair is a zone map, and it is almost free to store — one small summary per thousands of rows.

Watch what it buys. A query asks for keys between 4 and 7. Before reading anything, consult the zone maps: a partition whose range is [10, 20] cannot possibly contain a key in [4, 7], so skip it entirely — do not read one of its rows. A partition whose range is [3, 8] might, so scan it. The query reads only the partitions whose min–max band overlaps the query's band, and ignores the rest on the strength of a two-number summary. This is partition pruning, and its power scales with data: on a table partitioned into thousands of zones, a selective query touches a handful and prunes the rest, turning a full-table scan into a few targeted reads. The summaries did the filtering; the rows were never disturbed.

The technique depends on clustering — the rows being sorted or grouped by the key, so each partition's band is narrow. If the key values were scattered randomly across every partition, every zone map would span the whole range and nothing could be pruned. Clustering is what makes the summaries selective, and arranging

data so that its physical layout matches its query patterns is one of the quiet arts of database performance.

THE STORY

The idea of summarising a block of data so you can skip it is old and powerful, and US5710915 is a clean 1995 expression of it: cluster the rows, summarise each partition's range, and prune. Its importance exploded with the rise of columnar analytic databases and data warehouses, where tables grew to billions and then trillions of rows and full scans became unthinkable. The modern names are zone maps (Netezza, Oracle Exadata), min/max statistics and data skipping (the Parquet and ORC file formats, Apache Spark, Snowflake, every cloud warehouse). They are all the same move this patent describes: keep a cheap per-block summary of the values, and let a query consult the summaries to avoid reading blocks that cannot match. It is, arguably, the technique that makes "query a hundred-billion-row table interactively" possible at all — not a faster scan, but the systematic avoidance of scanning.

THE CODE

The query consults each partition's [min, max] and scans only the overlapping ones. Here is the pruned range query from patentml's `clustered_partition`:

```
def range(self, lo, hi):
    out = []
    self.scanned_partitions = 0
    for zlo, zhi, s, e in self._zones:      # each partition's min/max
band
        if zhi < lo or zlo > hi:
            continue                      # PRUNE: band can't overlap the
query
        self.scanned_partitions += 1
        for k, p in self.rows[s:e]:      # scan only the survivors
            if lo <= k <= hi:
                out.append((k, p if p is not None else k))
    return out
```

The demo clusters the keys 1–9 into partitions of three and asks for the range 4–7. It prints:

Clustered partition / zone map (from US5710915, expired):

```
keys 1..9 in partitions of 3
range 4..7 -> [4, 5, 6, 7]
scanned 2 of 3 partitions (pruned the rest)
```

The last line is the whole point. The keys clustered into three partitions — roughly [1,3], [4,6], [7,9] — and the query for 4–7 consulted their min/max bands, saw that [1,3] could not overlap, and

pruned it unread, scanning only the two partitions that might contain matches. On this toy it saved one partition of three; on a warehouse it is the difference between reading a trillion rows and reading a billion. The returned keys 4–7 are exactly correct. The module fuzz-tests four thousand datasets, checking the pruned range query against a full scan, and reports PASS — pruning never drops a real match.

THE INHERITANCE

The zone map is public domain, and it is one of the highest-leverage ideas in all of data engineering: a two-number summary per block that lets queries skip most of a table. Any time you store data in chunks and query it by a range — time-series by timestamp, logs by date, records by id — clustering the data on that key and keeping per-chunk min/max turns full scans into targeted reads, often by orders of magnitude. It is why the advice "sort your big table by the column you filter on" is worth real money: it makes every future zone map selective. The technique underpins every modern analytic file format and warehouse, and its 1995 core is now free — a reminder that the biggest speedups often come not from computing faster, but from arranging data so you can honestly refuse to compute at all.

CHAPTER 22

A LINE THAT FOLDS SPACE (THE Z-ORDER CURVE)

THE FILING

US Patent 5,781,906 — "System and method for construction of a data structure for indexing multidimensional objects."

Filed 6 June 1996. Expired 6 June 2016; public domain.

The grid of Chapter 20 handled space by dividing it. This chapter handles space by flattening it — mapping every two-dimensional (or n-dimensional) point onto a single number, in a way that keeps nearby points nearby, so that all the one-dimensional machinery of Part I suddenly works on spatial data. The claims describe constructing a single linear key from several coordinate fields so that objects with nearby coordinates get nearby keys and cluster together. The specific construction — interleaving the bits of the coordinates — is the Z-order or Morton curve, and it is one of the most elegant

ideas in the book.

THE IDEA

Here is the puzzle a grid does not solve: a B-tree indexes one sorted key beautifully, but space has two coordinates and no obvious single order. Sort points by longitude alone and two points on the same street but different blocks land far apart in the index. Is there a way to turn a two-dimensional point into one number such that points close in space get numbers close in value?

The Z-order curve does it with a trick so simple it feels like sleight of hand: interleave the bits of the coordinates. Take a point's x and y , write each in binary, and weave their bits together — first bit of x , first bit of y , second bit of x , second bit of y , and so on — into a single number, the Morton code. Points that share their high-order bits (they are in the same broad region of the plane) share the high-order bits of their Morton code, so they land near each other on the number line. The curve traces a recursive "Z" shape through space — hence the name — visiting all four quadrants, then recursing into each, so that locality in the plane becomes, mostly, locality along the line.

Once every point is a single Morton number, all of Part I applies. Store the codes in a B-tree or a sorted index; a point lookup is a key lookup; and a spatial box query becomes a small set of contiguous

range queries along the Morton line — because a rectangle in space maps to a few runs of nearby codes. You have indexed two-dimensional data with a one-dimensional structure, for free, by choosing the right key.

THE STORY

The Z-order curve was described by G. M. Morton at IBM in 1966, as a way to lay out a two-dimensional geographic database in linear file storage while keeping nearby places near each other on disk. It is genuinely foundational and long public-domain heritage — one of the small number of truly beautiful ideas in data structures. What patents like US5781906 industrialise, thirty years later, is the construction and use of such locality-preserving linear keys for indexing multidimensional objects inside real database systems: build the interleaved key, index it with the ordinary machinery, answer spatial queries as key ranges. Its descendants are pervasive and modern. Space-filling curves — Z-order and its smoother cousin the Hilbert curve — order the tiles of web maps, lay out spatial databases and geospatial file formats, cluster multidimensional data in analytic warehouses (Snowflake and Databricks both offer "Z-ordering" as a clustering option by that exact name), and index everything from astronomical sky surveys to game worlds. A 1966 idea for putting a map on a tape is, sixty years on, how the cloud clusters your multidimensional data.

THE CODE

The whole magic is the bit-interleave. Here is the Morton encoder from patentml's `zorder_curve`:

```
def morton_encode(coords, bits):
    code = 0
    for i in range(bits):          # for each bit position...
        for d, c in enumerate(coords):  # ...take that bit from each
coordinate
            code |= ((c >> i) & 1) << (i * len(coords) + d)
    return code
```

The demo encodes four points and then expands a spatial box into the cells it covers. It prints:

Z-order / Morton curve (from US5781906, expired):

```
encode (1,0),(0,1),(1,1),(3,3) b4 -> [1, 2, 3, 15]
  box (2,2)-(4,4) -> [(2, 2), (2, 3), (2, 4), (3, 2), (3, 3), (3, 4), (4, 2),
(4, 3), (4, 4)]
```

Read the encodings. The point (1,0) becomes Morton code 1, (0,1) becomes 2, (1,1) becomes 3 — the four points around the origin map to the small consecutive codes 1, 2, 3, tracing the "Z" through that corner of the plane. The point (3,3), off in the far corner, jumps to

15. Nearby points, nearby codes; distant point, distant code — exactly the locality that lets a B-tree index space. The box query then correctly enumerates the nine cells of the (2,2)–(4,4) rectangle, which in a real index would be fetched as a few contiguous Morton ranges. The module fuzz-tests the encoding, decoding, and box queries and reports PASS.

THE INHERITANCE

The Z-order curve is public domain, and it is the tool for indexing multidimensional data without a special multidimensional structure — when you would rather reuse the B-tree, the sorted index, the key-value store you already have. Interleave the coordinates into one key, and every one-dimensional tool indexes space; cluster your warehouse table by a Z-order of two columns and range-filters on both get fast at once. It is the most elegant demonstration in the book of a recurring lesson — that the right encoding can dissolve a hard problem into an easy one you already know how to solve. A point in the plane looked like it needed a whole new kind of index; interleave its bits and it is just a number on a line. That insight, and Morton's beautiful sixty-year-old curve, are yours to fold space with.

CHAPTER 23

A PLACE AS A NUMBER (GEOGRAPHIC LOCATION REFERENCING)

THE FILING

US Patent 5,839,088 — "Geographic location referencing system and method."

US Patent 6,047,236 — "Geographic location referencing system and method."

US Patent 6,202,023 — "Internet based geographic location referencing system and method."

Filed 1995–1997. All expired; public domain.

This trio of related filings solves a very particular and very practical problem: how do you describe where something is on a road network compactly, and resolve it back to a place, without transmitting raw latitude and longitude? Their claims describe encoding a location as a position along a known route — a distance from the route's start,

plus a lateral offset — so a place becomes a small pair of numbers that both ends can resolve against the same map. It is linear referencing, the way traffic reports, transport systems, and map databases pin a point to a road.

THE IDEA

Absolute coordinates — this latitude, this longitude — are precise but awkward. They are bulky to transmit, they say nothing about which road you mean, and a point that is a metre off can land in the wrong lane or the wrong field. For anything tied to a network of roads or rails, there is a more natural description: not "where on Earth" but "how far along this route."

Linear referencing encodes a location as a position along a route polyline. Given a route — a sequence of connected points forming a road — any nearby place is described by two numbers: the distance along the route to the closest point on it, and the lateral offset from the route to the actual place. To encode a point, you find the nearest point on the route (projecting onto each segment and keeping the closest), measure how far along the route that is, and note how far off to the side the original point sat. To decode a position back into coordinates, you walk along the route accumulating distance until you reach the given position, and interpolate. A place that sits exactly on the road has zero offset and round-trips perfectly; a place beside the road carries a small offset that says how far aside.

The payoff is compactness and meaning. "Route 12, 5.0 km along" is smaller than a coordinate pair, unambiguous about which road it refers to, and resolvable by anyone holding the same map — which is exactly what a traffic-message system, a fleet tracker, or a mapping database needs.

THE STORY

This family of patents comes from the late-1990s world of digital mapping and early telematics, when in-car navigation, digital road maps, and traffic-information services were being built and needed a common, compact language for place on a road. The problem was real: how does a traffic broadcast say "congestion here" in a way every receiver can locate, without sending precise coordinates that might not match each device's map exactly? Linear referencing — position along a route — was the answer, and it became a foundation of the technology. Its direct descendants include the location-referencing standards behind traffic-message services (TMC, and later OpenLR, an open standard for exactly this "encode a location so another map can find it" problem), the linear-referencing systems that transport and highway agencies use to record everything from potholes to speed limits by "milepost," and the map-matching that snaps a wandering GPS trace onto the road you are actually driving. Every "incident on the M6 northbound at junction 4" is a linear reference at heart.

THE CODE

Encoding projects a point onto the route and measures distance-along; decoding walks the route to a position. Here is encode from patentml's `geo_reference`:

```
def encode(self, point):                # -> (distance_along_route,
offset)
    best = None
    for i in range(len(self.route) - 1):
        t, foot, d = _seg_project(point, self.route[i], self.route[i + 1])
        s = self.cum[i] + t * (self.cum[i + 1] - self.cum[i]) # distance
along
        if best is None or d < best[0]:
            best = (d, s, foot)
    d, s, _ = best
    return s, d
```

The demo builds an L-shaped road and references a point against it. It prints:

Geo location referencing (from
US5839088/US6047236/US6202023, expired):

point (5,2) -> position 5.0 offset 2.0
decode(15) -> (10.0, 5.0)

The road runs from (0,0) to (10,0) to (10,10). The point (5,2) sits beside the first, horizontal segment: its nearest point on the road is (5,0), which is 5.0 units along the route, and it lies 2.0 units off to the side — exactly the encoding printed. Then `decode(15)` walks 15 units along the 20-unit road: 10 units gets to the corner (10,0), and the remaining 5 climb the vertical segment to (10, 5) — the coordinate returned. A place became two meaningful numbers and a position resolved back to a place, both against the same route. The module fuzz-tests five thousand random routes, confirming that a point exactly on the road encodes and decodes back to itself, and reports PASS.

THE INHERITANCE

Geographic location referencing is public domain, and it is the tool for any application that lives on a network of routes rather than open space: fleet and asset tracking by road position, transport and logistics systems, anything that records events "along a line" (a railway, a pipeline, a delivery route, a river), and the map-matching that turns noisy GPS into a clean position on a known road. Linear referencing is smaller than coordinates, robust to small errors, and — crucially — shareable: two systems with the same base map can exchange locations as route-and-offset and agree on where they mean. It is a beautiful example of choosing coordinates that fit the problem: on a road network, "how far along" is simply a better address than "where on Earth," and this trio of expired filings hands

you the machinery to use it.

CHAPTER 24

SORTING THE WORLD INTO REGIONS (GEOGRAPHIC CLASSIFICATION)

THE FILING

US Patent 6,148,289 — "System and method for geographically organizing and classifying businesses on the world-wide web."

Filed 1996. Expired; public domain.

The last chapter pinned points to roads. This one organises points by area — bucketing places into grid cells for fast spatial grouping, and classifying each into the nearest named region. The claims describe organising records by location: quantising each point to a grid cell for fast bucketing, and assigning it to the nearest region centroid. It is the machinery behind the local directory — the "businesses near you," the "results in your area" — and behind any system that has to answer "which region does this belong to?" at scale.

THE IDEA

This chapter is a fusion of two ideas you have already met, applied to geography. From the grid index of Chapter 20 it takes spatial bucketing: quantise each point's coordinates to a cell key (floor the latitude and longitude by a cell size), so all the places in the same neighbourhood share a key and can be gathered instantly. From the world of clustering it takes nearest-centroid classification: given a set of named regions, each with a representative centre, assign any point to the region whose centre is closest.

Together they answer the two questions a local directory must answer constantly. "What else is right here?" is a bucket lookup — hash the point to its cell and return that cell's members, no scan. "What region is this in?" is a nearest-centroid check — compare the point to each region's centre and pick the closest. The first groups places by proximity; the second labels them by area. A business at a location is filed into its grid cell (so "near here" is fast) and tagged with its region (so "results in the north" is a filter). The whole apparatus of local search — group by nearness, classify by region — falls out of quantising space into cells and measuring distance to region centres.

THE STORY

US6148289 is a artifact of a very specific moment: the late-1990s

scramble to organise the physical world of businesses onto the world-wide web — the birth of the online local directory, the digital yellow pages, "find shops near you." The problem it addresses is exactly the one every local-search product since has faced: given millions of businesses with locations, how do you organise them so a user can browse by area and find what is nearby, quickly? Its answer — grid-bucket for proximity, nearest-region for classification — became the skeleton of local search. Its descendants are the "restaurants near me," the store locators, the regional filters on every marketplace, the geofencing that fires when you enter an area, and the geospatial bucketing (geohashes, H3 hexagons, S2 cells) that every mapping platform uses to group and serve places by location. Whenever an app shows you "what's around here" or "results in your city," a geographic bucketing-and-classification scheme is underneath it.

THE CODE

Quantise to a cell for bucketing; compare to region centres for classification. Here are both from `patentml`'s `geo_classify`:

```
def cell_key(point, size):          # spatial bucket key
    return (int(math.floor(point[0] / size)), int(math.floor(point[1] /
size)))

def classify(self, point):          # nearest named region
```

```
return min(self.regions.items(),  
           key=lambda kv: math.dist(point, kv[1]))[0]
```

The demo files three shops by location and classifies two points against a north/south split. It prints:

Geographic classification (from US6148289, expired):

```
cell of (1,8): (0, 1) -> members: ['shop1', 'shop2']  
  classify (0,6) -> north  
  classify (0,-6) -> south
```

The first line shows spatial bucketing at work: shop1 at (1,8) and shop2 at (2,9) both quantise to grid cell (0,1) — they are near each other, so they share a bucket, and "what's near shop1?" returns them together without scanning the other shops. The classifications then label points by region: (0,6) is closer to the north centre than the south, so it is classified north; (0,-6) goes south. Grouping by nearness and labelling by area, both from simple geometry. The module fuzz-tests four thousand cases, checking nearest-region classification against brute force and confirming the cell-membership invariant, and reports PASS.

THE INHERITANCE

Geographic classification is public domain, and it is the tool for

organising anything by where it is: local directories and store locators, "near me" search, regional filtering and reporting, geofencing, and any dataset you want to bucket and browse by area. Grid-bucketing gives you instant "what's nearby" without a spatial scan; nearest-centroid gives you clean regional labels; together they are the backbone of local search, in a form small enough to run on the device asking the question. It closes Part V by bringing space full circle — from indexing points (Chapter 20), to pruning by region (21), to folding space onto a line (22), to pinning places to routes (23), and finally to sorting the whole world into browsable regions. Data with coordinates, in all its forms, is now in your hands. Part VI turns from finding data to storing it: the allocation maps, caches, and locks that manage the bytes themselves.

PART VI — STORAGE AND SYSTEMS

Every structure in this book has to live somewhere — on disk, in memory, across a cluster of machines. This part drops beneath the query and the index to the plumbing they rest on: how a store decides which blocks are free to write into, how a system remembers results so it need not recompute them, how a fast small cache is kept in step with a larger slow one, and how a cluster of machines agrees on who is allowed to write to a shared file. These are the unglamorous algorithms that never appear in a query plan and without which nothing else in the book could run. Four chapters on managing the bytes themselves.

CHAPTER 25

WHERE THE BYTES LIVE (THE ALLOCATION MAP)

THE FILING

US Patent 5,960,446 — "Parallel file system and method with allocation map."

Filed 1997. Expired; public domain.

Before a database can write a row or a file system can save a file, something must decide where the bytes go — which blocks of the underlying store are free, and which are already taken. This patent's claims describe tracking that with a bitmap: one bit per block, on for used and off for free, so that allocation is a search for a run of free bits and freeing is a flip back to zero. It is the free-space map, the humble accountant that sits under every file system and storage engine.

THE IDEA

A storage device is a long row of fixed-size blocks. To store something you need free blocks; to store something contiguous you need a run of adjacent free blocks. The problem is bookkeeping: which blocks are free right now, and where is there a big enough gap?

The allocation map answers with the cheapest possible representation — a bitmap, one bit per block, 1 for used and 0 for free. Allocation of n blocks is a scan for the first run of n consecutive zero bits (a first-fit search); once found, those bits are set to 1 and their starting position returned. Freeing is simply flipping the relevant bits back to 0, returning the space to the pool. Two more questions fall out for free: how much space is left (count the zeros) and what is the largest thing you could still store contiguously (find the longest run of zeros). The whole free-space accounting of a storage system reduces to setting, clearing, and scanning bits — compact enough that the map for an enormous store fits in a sliver of memory.

The interesting behaviour is fragmentation. Allocate and free in a mixed pattern and the free space breaks into scattered holes; a later request for a large contiguous run may fail even though the total free space is ample, because it is spread across gaps too small to use. The allocation map makes this visible — the largest-free-run query is

exactly "how fragmented am I?" — and first-fit's habit of reusing the earliest gap is one simple strategy for keeping fragmentation in check.

THE STORY

Free-space bitmaps are foundational operating-systems and file-system technology, used since the early days of managing disks. US5960446's setting is a parallel file system — many machines writing to shared storage at once — where allocation must be fast and correct under concurrent pressure, and a compact, easily-locked bitmap is exactly the right structure. The idea is everywhere bytes are managed: the block allocators of file systems (ext, NTFS, and countless others track free blocks with bitmaps), the page allocators inside operating-system kernels and memory managers, the space maps of databases deciding where to put the next page, and the free-list management of storage arrays. It is one of those algorithms so fundamental that it is easy to forget it is an algorithm at all — until you watch a disk fail to save a large file despite showing gigabytes free, and realise the allocation map is telling you the space is there but not together.

THE CODE

Allocation is a first-fit scan for a run of free bits; freeing flips them back. Here is alloc from patentml's `allocation_map`:

```
def alloc(self, count=1):
    run = 0
    for i in range(self.n):
        if self.used[i]:
            run = 0                # occupied block breaks the run
        else:
            run += 1
            if run == count:        # found count consecutive free blocks
                start = i - count + 1
                for b in range(start, start + count):
                    self.used[b] = 1
                return start
    return None                    # no gap large enough
```

The demo allocates three chunks, frees the middle one, and reallocates into the hole. It prints:

Allocation map (from US5960446, expired):

```
alloc 4,4,2 -> starts 0 4 8 | free: 6
free middle 4 -> largest contiguous run: 6
alloc 3 -> start 4 (fits in the freed hole)
```

Trace the fragmentation story. Three allocations of 4, 4, and 2 blocks filled positions 0–9 of a 16-block store, leaving 6 free at the end. Freeing the middle chunk (blocks 4–7) returned four blocks to the

pool — and now the largest contiguous run is 6 (the untouched tail), even though total free space is 10, because the free space is split between the reopened hole and the tail. The final alloc 3 then reused the freed hole, starting at block 4 — first-fit chose the earliest gap that fit. Every step is bit-setting and scanning. The module fuzz-tests three thousand allocate/free sequences, checking first-fit placement and the free count against an independent model, and reports PASS.

THE INHERITANCE

The allocation map is public domain, and it is the tool whenever you manage a pool of fixed-size resources and need to hand out and reclaim contiguous runs: blocks on a disk, pages in memory, slots in a buffer, frames in a ring, chunks of any array-backed arena. A bitmap, a first-fit scan, and a free-flip give you a complete, compact allocator in a couple of dozen lines — and the largest-free-run query hands you a fragmentation gauge for nothing. More than most chapters, this one demystifies something you rely on invisibly every day: the reason your files save, and the reason they sometimes cannot despite free space, is a stripe of bits being scanned and flipped exactly like this. Now that this 1997 filing has expired, that stripe of bits is yours to build storage on.

CHAPTER 26

REMEMBERING BY CONTENT (THE CONTENT-ADDRESSABLE CACHE)

THE FILING

US Patent 6,292,880 — "Alias-free content-indexed object cache." Filed 1998. Expired; public domain.

Most caches remember things by name: "what is stored under key K?" This one remembers by content: "have I already stored this exact data, whatever it was called?" The claims describe indexing cached objects by a hash of their content rather than by name, so that two objects with identical content share a single stored copy — no duplicates, no matter how many names point at them. It is content addressing, the idea behind deduplicating storage, and behind the version-control systems that hold much of the world's code.

THE IDEA

A normal cache is a map from names to data: store a.txt, store copy.txt, and even if they hold identical bytes you now keep two copies, because they have two names. That is wasteful whenever the same content arrives under different names — which, across a real system, it does constantly.

A content-addressable cache inverts the addressing. Instead of "store this under this name," it computes a hash of the content itself and uses that as the key. Identical content produces an identical hash, so two objects with the same bytes map to the same slot and are stored only once; a hundred names for the same data cost one copy plus a hundred tiny pointers. Retrieval by name works by keeping a small `name!hash table` alongside the content store, but `table` keyed by what the data is, not what it is called. The "alias-free" in the patent's title is the promise: no aliases, no duplicate copies of identical content, ever.

Two lovely properties fall out. Automatic deduplication: sameness is detected for free, because identical content simply lands on the same key — you never search for duplicates, they cannot exist. And integrity: because the key is the hash of the content, you can verify at any time that stored data has not been corrupted or tampered with, by rehashing it and checking that it still matches its address. The name is a label; the hash is the truth.

THE STORY

Content addressing is one of the deep ideas of computing, and by the late 1990s it was moving into storage systems as data volumes made duplication expensive. US6292880's "alias-free content-indexed object cache" is a filing from that moment — cache and store objects by content hash so identical objects coalesce. The idea went on to reshape whole categories of software. Deduplicating storage and backup systems store each unique block once and reference it many times, often shrinking backups by an order of magnitude.

Content-addressable storage systems keep data by its fingerprint for integrity and immutability. Most visibly, Git — the version-control system holding a vast share of the world's source code — is a content-addressable store at its core: every file and every commit is keyed by the hash of its content, which is exactly why Git can tell instantly whether two versions are identical and why it never stores the same content twice. The peer-to-peer web (IPFS) addresses files the same way. A modest 1998 cache patent sits in the lineage of one of the most important architectural ideas in modern software.

THE CODE

The store is keyed by a hash of the content; a name table maps names to those hashes. Here is the put from patentml's `content_addressable_cache`:

```
def put(self, name, content):  
    key = self._hash(content)      # address = hash of the
```

CONTENT

```
if key not in self.store:
    self.store[key] = content    # store identical content only once
self.names[name] = key        # the name just points at the hash
return key
```

The demo stores three names holding two distinct contents. It prints:

Content-addressable cache (from US6292880, expired):

```
put 3 names, 2 distinct contents -> {'names': 3, 'unique_blobs': 2,
'dedup_saved': 1}
a.txt and copy.txt share key: True
get('copy.txt') -> b'hello world'
```

The numbers tell the deduplication story exactly. Three names were stored, but only 2 unique blobs were kept — one copy was saved, because two of the names (a.txt and copy.txt) held identical content and therefore hashed to the same key and shared a single stored copy. The line a.txt and copy.txt share key: True confirms it: same content, same address. Yet retrieval by name still works — get('copy.txt') returns its bytes — because the name table quietly redirects to the shared blob. Three names, two copies, zero duplicate storage. The module verifies the deduplication and retrieval and reports PASS.

THE INHERITANCE

The content-addressable cache is public domain, and it is the tool for any system where the same data arrives under many names or in many places: deduplicating storage and backups, immutable object stores, content delivery, asset pipelines, anywhere you would otherwise keep needless copies of identical bytes. Key by content hash and duplication becomes impossible rather than merely discouraged, integrity checking becomes free, and equality becomes a hash comparison instead of a byte-by-byte scan. It is also the gateway to understanding Git and the content-addressed web — once you see that addressing data by what it is rather than what it is called dissolves both duplication and tampering in one move, a whole family of modern systems suddenly makes sense. That idea, and a working content-addressable cache, are yours now that this filing has expired.

CHAPTER 27

TWO CACHES, KEPT IN STEP (THE TWO-LEVEL CACHE)

THE FILING

US Patent 6,825,848 — "Synchronized two-level graphics processing cache."

Filed 2000. Expired; public domain.

Caches trade size against speed: a small cache is fast but holds little, a large cache holds much but is slower. Why choose? This patent's claims describe a hierarchy — a small fast first level backed by a larger slower second level, kept synchronized so that everything in the fast cache is also in the slow one. A lookup checks the fast level first, then the slow; a miss fetches and fills both. It is the cache hierarchy, the same idea that gives your processor its L1, L2, and L3 caches, applied here to a graphics pipeline.

THE IDEA

You cannot have a cache that is both huge and instant — physics and cost forbid it. But you can stack caches to get most of both. Keep a small, fast first-level cache (L1) and a larger, slower second-level cache (L2) behind it, and look things up in order: check L1; if it is there, you are done at top speed. If not, check L2; if it is there, you found it more slowly but still faster than the original source, and you promote it into L1 so the next access is fast. If it is in neither — a true miss — you fetch it from the slow underlying source and fill it into both levels.

The discipline that keeps this honest is inclusion: everything in L1 must also be in L2. The fast cache is a subset of the slow one, a hot working set skimmed off the top. This is what "synchronized" means in the patent's title — the two levels are not independent caches that might disagree, but a coherent hierarchy where L1 holds the hottest items and L2 holds those plus the merely-warm ones. Each level manages its own capacity by evicting its least-recently-used item when full; because L1 is smaller, items fall out of L1 sooner (back to L2) than they fall out of the system entirely (evicted from L2). The result behaves like a single cache that is both large and fast, because the items you touch most stay in the fast level while the larger level catches the rest.

THE STORY

Cache hierarchies are one of the most important ideas in computer architecture — the entire memory system of a modern computer is a hierarchy, from registers to L1/L2/L3 caches to RAM to disk, each level larger and slower than the one above, exploiting the fact that programs reuse a small hot set of data intensely. US6825848 applies that principle to a graphics processing pipeline, where texture and vertex data must be served at enormous rates and a synchronized two-level cache keeps the hottest data closest to the processor. The pattern generalises far beyond graphics or CPUs. It is the architecture of web caching (browser cache in front of a CDN in front of the origin), of database buffer pools layered over disk, of the memory tiers in every serious storage system, and of application caches (a small in-process cache backed by a larger shared one like Redis). Whenever you see "L1/L2," "hot/warm/cold tiers," or "near cache / far cache," this chapter's idea is at work: stack caches by speed, keep them inclusive, and let the hottest data float to the top.

THE CODE

Look up L1, then L2; an L2 hit promotes into L1; each level evicts its least-recently-used when full. Here is get from patentml's `two_level_cache`:

```
def get(self, key):  
    if key in self.l1:  
        self.l1.move_to_end(key); self.l2.move_to_end(key)
```

```

    return "L1"                # fast-path hit
if key in self.l2:
    self.l2.move_to_end(key)
    self._promote(key, self.l2[key]) # L2 hit -> pull into L1
    return "L2"
return None                    # true miss -> caller fills both

```

The demo uses a 2-slot L1 behind a 4-slot L2, fills it with four items, then probes. It prints:

Two-level inclusive cache (from US6825848, expired):

```

after put a,b,c,d -> L1: ['c', 'd'] L2: ['a', 'b', 'c', 'd']
get('a') -> L2 (pushed out of small L1, still in L2 -> L2 hit,
promoted)
get('d') -> L1 (still in fast L1)
inclusive invariant holds: True

```

Read the hierarchy at work. After putting a, b, c, d, the small L1 holds only the two most-recent (c, d), while the larger L2 holds all four — and L1 is a subset of L2, as inclusion requires. Asking for a finds it not in the fast L1 (it was pushed out when c and d arrived) but still in L2 — an L2 hit, slower than L1 but far faster than refetching, and a is promoted back into L1 for next time. Asking for d hits the fast L1 directly. The inclusion invariant holds throughout. The module fuzz-tests four thousand workloads with random cache

sizes, checking the inclusion and capacity invariants after every operation, and reports PASS.

THE INHERITANCE

The two-level cache is public domain, and its lesson is one of the most broadly applicable in all of computing: you rarely have to choose between fast and big — you can stack them. A small fast cache in front of a larger slow one, kept inclusive, behaves like a cache that is both, because access patterns are almost never uniform and the hot set is almost always small. Reach for it whenever a single cache forces an uncomfortable size/speed trade-off: an in-process cache over a shared one, a memory tier over disk, a local cache over a remote service. Understanding it also demystifies the machine you are reading this on — the L1/L2/L3 in your processor's spec sheet is exactly this pattern, and the reason your code runs fast is that its hot data floats to the top of a hierarchy just like this one. That architecture, and a working two-level cache, are yours now that this 2000 filing has expired.

CHAPTER 28

ONE WRITER AT A TIME (THE CLUSTER LOCK MANAGER)

THE FILING

US Patent 5,828,876 — "File system for a clustered processing system."

Filed 1996. Expired; public domain.

When one machine owns a file, keeping it consistent is easy — there is only one writer. When many machines share the same files, chaos threatens: two nodes writing the same file at once can corrupt it and leave their caches disagreeing about its contents. This patent's claims describe a clustered file system that hands out an exclusive ownership token per file: acquiring the token on one node revokes it from whoever held it, guaranteeing a single writer at a time. It is the distributed lock manager, the referee that lets a cluster share data without tearing it apart.

THE IDEA

The problem is coordination without chaos. Several machines, a shared pool of files, and a hard rule: no two machines may write the same file at the same moment, or the file — and the caches each machine keeps of it — will end up inconsistent. You need a referee that everyone obeys.

The lock manager is that referee, and its mechanism is the exclusive token. For each file there is exactly one token, and only the machine holding it may write. When a machine wants to write a file, it acquires the token; if another machine currently holds it, the manager revokes it from that machine first — telling it to stop writing and flush its changes — and then grants it to the requester. Because the token is single and its handoff is mediated, there is never a moment when two machines both believe they may write. Releasing the token returns the file to an unowned state, free for the next acquirer. The single most important guarantee is the invariant: at every instant, a file has at most one owner. That one rule, enforced on every acquire, is what makes shared storage safe.

The revocation is the subtle, valuable part. Acquiring is not just "take the token" — it is "take the token from whoever has it," which requires telling the previous owner to relinquish it cleanly. That handoff is where cache coherence lives: the old owner writes back its pending changes before letting go, so the new owner sees an

up-to-date file. Ownership and consistency are two faces of the same token.

THE STORY

Clustered and shared-disk file systems were a major line of systems research and engineering through the 1990s, as organisations built clusters of machines that needed to share storage for performance and availability. US5828876 addresses the coordination problem at their heart: how does a cluster of independent nodes share files without corrupting them? The distributed lock manager — token ownership with revocation — is the classic answer, and it is foundational infrastructure. Its descendants run under shared-disk cluster file systems (GFS2, OCFS2, VMware's VMFS), under clustered databases like Oracle RAC where nodes coordinate access to shared data blocks, and, in generalised form, under the distributed coordination services (ZooKeeper, etcd, and the consensus systems behind them) that modern distributed systems use to elect leaders, grant leases, and guard shared resources. The vocabulary grew richer — leases, fencing, consensus — but the core is this patent's: a single token per resource, handed off under a referee, so that the cluster agrees on who may act.

THE CODE

Acquiring grants the token and revokes it from the previous holder;

the single-owner invariant is the guarantee. Here is acquire from patentml's clustered_fs:

```
def acquire(self, file, node):
    prev = self._owner.get(file)
    if prev == node:
        return None                # already the owner; nothing revoked
    if prev is not None:
        self.revocations.append((file, prev, node)) # revoke from the
old owner
    self._owner[file] = node        # grant to the new owner
    return prev                    # who it was taken from
```

The demo has two nodes contend for one file. It prints:

Clustered file lock manager (from US5828876, expired):

```
node1 acquire fileA -> revoked from: None
node2 acquire fileA -> revoked from: node1
owner of A      : node2
node1 still holds A : False
```

Watch the token change hands. node1 acquires fileA when it is unowned — nothing to revoke, so "revoked from: None," and node1 may now write. When node2 acquires the same file, the manager revokes it from node1 first ("revoked from: node1") and grants it to

node2. Afterwards the owner is unambiguously node2, and — the crucial line — node1 still holds A: False. There is never a moment when both believe they may write; the token moved cleanly from one to the other. The module fuzz-tests five thousand acquire/release sequences across three nodes, checking the single-owner invariant against an independent model after every operation, and reports PASS.

THE INHERITANCE

The cluster lock manager is public domain, and its idea is the seed of all distributed coordination: to let many actors share a resource safely, funnel access through a single token handed off under a referee, and hold the invariant that at most one holds it at a time. You meet this pattern the instant more than one process, thread, or machine must take turns at a shared thing — a file, a database row, a job queue, a leadership role. The modern tools dress it up as leases, mutexes, and consensus, but the heart is this 1996 filing's exclusive token with clean revocation. Understanding it demystifies the whole anxious vocabulary of concurrency and distribution: the fear is always "two writers at once," and the answer is always "one token, one owner, handed off carefully." That answer, and a working lock manager to build on, close Part VI — the plumbing of storage and coordination that everything else in this book quietly stands on. Part VII turns outward one last time, to the algorithms that use all of this to connect data back to the people who want it.

PART VII — DATA THAT FINDS PEOPLE

Every part so far has been about storing data and answering questions put to it. This final part turns the relationship around: instead of a person querying the data, the data reaches out to find the person. These are the algorithms of discovery and recommendation — matching people by shared interest, grouping users and items together, finding content by example rather than by keyword, laying clusters out to be seen, comparing series that move differently, and encoding a link into something you can print and scan. They are where a database stops being a filing cabinet and becomes a matchmaker. Six chapters to close the book, on the algorithms that turn stored data into connection.

CHAPTER 29

WHO ELSE LIKES THIS (INTEREST MATCHING)

THE FILING

US Patent 5,796,395 — "System for publishing and searching interests of individuals."

Filed 1996. Expired; public domain.

Long before "people you may know" and "recommended for you," someone had to invent the basic move: represent what a person is interested in, and find the other people whose interests are most alike. This patent's claims describe exactly that — individuals publish interest profiles, and a query profile is matched against them by similarity so the most-aligned people surface first. It is the seed of the interest graph, the machinery of discovery that now underlies every social and recommendation system.

THE IDEA

To match people by interest, you first need to turn "interest" into something a computer can compare. The move — which you have already met in Chapter 12's search and Chapter 15's matching — is the vector: represent each person's interests as a bag of weighted terms, a point in a space with one axis per possible interest. Someone into "python machine learning data science" becomes a vector leaning in those directions; someone into "guitar jazz blues" leans elsewhere entirely.

Now similarity has a geometric meaning. Two people are alike to the degree their vectors point the same way, measured by cosine similarity — the cosine of the angle between them, which is 1.0 when they point identically, 0 when they are unrelated, regardless of how many interests each has. To find who is like you, compute the cosine between your profile and everyone else's, keep those above a threshold, and rank them by score. The people whose interests align most closely with yours rise to the top. That single operation — profile as vector, match by cosine, rank by score — is the atom of the entire discovery economy: "who is like this person?", "what is like this thing?", "who should meet whom?"

THE STORY

US5796395 is a remarkably early filing — 1996, before the social

web existed in any recognisable form — that anticipates its central mechanic: publish what you care about, and let the system connect you to others who care about the same things. It comes from the moment the internet was first imagined as a place to find your people, and its interest-profile-and-match idea is the direct ancestor of the interest graph that social platforms are built on. Its descendants are everywhere connection is suggested: "people you may know," "accounts to follow," dating-app matching, "readers who liked this," professional-network suggestions, and the content recommendation that pairs items to the users most likely to want them. All of them, underneath, represent people (and things) as vectors and match by similarity — exactly this chapter. The interest graph turned out to be one of the most valuable structures ever built, and here is its unglamorous 1996 seed.

THE CODE

Profiles become term vectors; matching is cosine similarity above a threshold, ranked. Here is the matcher from patentml's `interest_match`:

```
def cosine(a, b):
    common = set(a) & set(b)
    dot = sum(a[t] * b[t] for t in common)
    na = math.sqrt(sum(v * v for v in a.values()))
    nb = math.sqrt(sum(v * v for v in b.values()))
```

```
    return dot / (na * nb) if na and nb else 0.0

def match(self, query, threshold=0.1, k=None):
    q = profile(query)
    scored = [(who, round(cosine(q, p), 4)) for who, p in
self._people.items()]
    scored = [(w, s) for w, s in scored if s >= threshold]
    scored.sort(key=lambda t: (-t[1], repr(t[0])))
    return scored[:k] if k else scored
```

The demo publishes three people's interests and searches for someone interested in "python data engineering." It prints:

Interest matching (from US5796395, expired):

```
ana  score=0.5164
cat  score=0.5164
```

The query for "python data engineering" matched ana (published: "python machine learning data science") and cat (published: "python web django backend data") — the two people whose interests overlap the query on python and data — while ben ("guitar music jazz blues"), who shares nothing, fell below the threshold and never appears. Both matches score identically because each shares two of the query's terms in a comparable way, and they are ranked together. Nobody was scanned exhaustively for meaning; interests became

vectors and alignment became a cosine. The module verifies that an exact-profile match tops the ranking at cosine 1.0 and that disjoint interests are filtered out, reporting PASS.

THE INHERITANCE

Interest matching is public domain, and it is the starting point for any discovery or recommendation feature: connecting users by shared interest, suggesting people or content, building a simple "more like this" over profiles. Represent each entity as a vector of what it is about, match by cosine, rank by score — and you have real, explainable recommendation in a handful of lines, running locally over data that need never leave your control. It is worth pausing on that last point: the interest graph is among the most powerful and most extracted structures of the modern web, built by platforms that mine your interests to sell your attention. Its foundational algorithm, though, is small, transparent, and now free — which means the same matchmaking that powers billion-dollar feeds can be built by anyone, on anyone's terms, for a community that owns its own graph. That possibility, and a working interest matcher, are yours now that this 1996 filing has expired.

CHAPTER 30

GROUPING BOTH WAYS AT ONCE (CO-CLUSTERING)

THE FILING

US Patent 7,072,846 — "Clusters for rapid artist-audience matching."

Filed 1999. Expired; public domain.

Ordinary clustering groups things of one kind — customers into segments, say. But recommendation data comes as a grid: users down the side, items across the top, ratings in the cells. There is structure in both directions at once — groups of similar users and groups of similar items — and they reinforce each other. This patent's claims describe clustering both simultaneously, grouping users and items together so that each block of the grid is roughly uniform, and recommendations flow from a user's block. It is co-clustering, and it is a beautifully symmetric idea.

THE IDEA

Picture a table of ratings: each row is a user, each column an item (a song, a film, a product), each cell how much that user liked that item. Cluster the rows alone and you find user segments; cluster the columns alone and you find item genres. But the two are entangled — a segment of users is defined by the genres they favour, and a genre is defined by the users who favour it. Clustering one at a time ignores that mutual dependence.

Co-clustering groups both at once. It seeks a partition of users into row-clusters and items into column-clusters such that each resulting block — the intersection of a user-group and an item-group — is as uniform as possible: everyone in that user-group feels roughly the same about everything in that item-group. The algorithm alternates, like a two-sided version of k-means: hold the item-groups fixed and assign each user to the user-group whose blocks best match their ratings; then hold the user-groups fixed and reassign each item; repeat until it settles. When it converges, the grid has been reordered into clean rectangular blocks of agreement — "these users love these items, and are indifferent to those." A recommendation for a user is then read straight off their block: the item-groups their user-group rates highly. Users and items are understood together, because in recommendation they were never really separate.

Like k-means, it can settle into a poor local arrangement depending

on where it starts, so patentml's module runs several random restarts and keeps the tightest result — the one whose blocks are most uniform.

THE STORY

US7072846's framing — "rapid artist-audience matching" — places it squarely in the dawn of digital music and media recommendation around the turn of the millennium, when services needed to connect artists to the audiences most likely to love them, at scale, from sparse listening data. Co-clustering was the natural tool: find the groups of listeners and the groups of artists simultaneously, because a listener-segment and an artist-cluster define each other. The idea belongs to a rich lineage of collaborative filtering and matrix-structure methods that power recommendation to this day. Its relatives — biclustering in bioinformatics (finding groups of genes that behave alike across groups of conditions), block models in network analysis, and the matrix-factorisation methods that won the famous Netflix Prize — all share co-clustering's core insight: when your data is a grid of interactions, the structure lives in both dimensions at once, and finding it in both is stronger than finding it in either alone. Every "listeners who like this artist also like..." carries this idea's DNA.

THE CODE

Alternate: assign rows to the best-matching row-cluster given the blocks, then columns, until stable; keep the best of several restarts. Here is the block-mean and the objective that guides it, from patentml's coclustering:

```
def _block_mean(M, rows, cols):  
    if not rows or not cols:  
        return 0.0  
    return sum(M[i][j] for i in rows for j in cols) / (len(rows) *  
len(cols))
```

each row is reassigned to the row-cluster whose blocks best predict its ratings;
then each column likewise; the objective is total squared error from block means,
minimised across several random restarts.

The demo co-clusters a 4×4 ratings grid with a clean 2×2 block structure. It prints:

Co-clustering (from US7072846, expired):

row clusters: [1, 1, 0, 0]

col clusters: [1, 1, 0, 0]

users {0,1} share a cluster: True | {2,3} share: True

The grid was built so that users 0–1 love items 0–1 and are cold on items 2–3, while users 2–3 are the mirror image. Co-clustering recovered exactly that: users 0 and 1 landed in the same row-cluster, users 2 and 3 in the other, and the items split the same way — the four blocks of agreement found simultaneously from both sides. A recommendation for user 0 now reads off their block: the item-group that user-cluster rates highly. The module fuzz-tests four hundred randomly-shuffled block matrices, checking that it recovers the planted row and column groups, and reports PASS.

THE INHERITANCE

Co-clustering is public domain, and it is the tool for any data shaped like a grid of interactions between two kinds of thing: users and items, readers and articles, customers and products, genes and conditions, words and documents. Grouping both dimensions at once finds structure that one-sided clustering misses, and turns a sparse, bewildering matrix into a handful of clean blocks you can act on and explain. It is also a small lesson in symmetry — that when two things define each other, you should solve for them together rather than pretending one comes first. That principle, and a working co-clustering engine, are yours now that this filing has expired — the recommendation core of the media age, small enough to run on the data of a single community that would rather match its own artists to its own audience than rent the service from a platform.

CHAPTER 31

SEARCH BY EXAMPLE (CONTENT-BASED RETRIEVAL OF IMAGES AND VIDEO)

THE FILING

US Patent 5,539,427 / US Patent 5,867,150 — "Graphic indexing system."

US Patent 5,802,361 — "Method and system for searching graphic images and videos."

US Patent 6,631,522 — "Method and system for indexing, sorting, and displaying a video database."

US Patent 6,389,168 — "Object-based parsing and indexing of compressed video streams."

Filed 1994–1998. All expired; public domain.

Text search (Chapter 4) needs words. But how do you search images and video, which have no words — only pixels? This cluster of filings answers: reduce each image or clip to a feature signature,

index the signatures, and search by example — "find me more that look like this one." Their claims describe indexing media by extracted features and retrieving by similarity, and parsing video streams into indexable objects. It is content-based retrieval, the way you search a haystack of pictures with a picture.

THE IDEA

Searching media by example rests on the same move that ran through Chapters 12, 15, and 29: turn the thing into a vector, then compare vectors. For an image, the vector is a feature signature — a compact numeric summary of its visual content, such as a coarse grid of average intensities (patentml's `graphic_index` uses exactly this: downsample the image to a small normalised grid). Similar images produce similar signatures; to find images like a query, compute its signature and return the stored images whose signatures are nearest. Keyword search became example search: the query is not words but a picture, and "relevance" is distance in feature space.

Video adds two twists, each covered by its own filing. First, a clip is not one image but a sequence, so it is summarised by the feature vectors of its keyframes, and a clip matches a query if any of its keyframes is close to the query — patentml's `media_index` measures exactly this minimum cross-frame distance. Second, video arrives as a compressed stream of marker-delimited units, so before you can index it you must parse it into objects (keyframes, motion frames,

audio) — patentml's `video_object_index` splits a marked stream into typed objects and indexes them by type and time, so you can jump to "the keyframes between second one and second three" without decoding the whole stream. Signature, keyframe, parsed object: three refinements of one idea — index media by what it contains, and search it by example.

THE STORY

Content-based image and video retrieval was a hot and hard problem of the 1990s, as digital media collections exploded and the old approach — hand-typing keywords onto every image — collapsed under the volume. Systems like IBM's QBIC ("Query By Image Content") pioneered searching by colour, texture, and shape rather than by caption, and this family of patents comes from that wave: index the visual features, search by example, parse video into indexable structure. The lineage runs straight to the present. Reverse image search ("find this picture on the web"), the "similar images" and "similar products" on shopping sites, near-duplicate video detection, content-ID systems that match uploads against a catalogue, and the visual search in photo libraries all descend from it — and the modern versions simply swap the hand-designed signature for a neural-network embedding, while keeping the architecture identical: media becomes a vector, and search becomes nearest-neighbour by example. The signatures got smarter; the idea in these filings did not change.

THE CODE

An image becomes a normalised feature signature; nearest signature wins. Here is signature from patentml's `graphic_index`:

```
def signature(image, cells=4):
    # downsample the image to a cells x cells grid of average
    intensities,
    # then L2-normalise -> a compact vector where near = visually
    similar

    ...
    norm = math.sqrt(sum(v * v for v in sig)) or 1.0
    return tuple(v / norm for v in sig)
```

Three demos show the three refinements. Image search:

Graphic indexing (from US5539427/US5867150, expired):

query image 3 -> nearest: (3, 0.0) (should be itself, distance ~0)

Video-clip search by nearest keyframe:

Media indexing (from US5802361/US6631522, expired):

query near (0,0) -> ranked: [('clipC', 0.141421), ('clipA', 0.223607), ('clipB', 6.8593)]

And parsing a compressed stream into indexed objects:

Object-based stream indexing (from US6389168, expired):

```
keyframes      : ['I0', 'I1', 'I2']  
keyframes t in 1..3 -> ['I1']
```

Together they tell the whole story. Querying the image index with image 3 returned image 3 itself at distance 0.0 — a signature always retrieves its own image, the property that makes the index trustworthy. The video search ranked the clips by their nearest keyframe to the query point near (0,0): clipC first (it has a keyframe at distance 0.14), then clipA, then the distant clipB — retrieval by visual example across sequences. And the stream index parsed a compressed video into typed objects and answered "keyframes between t=1 and t=3" with just I1, jumping to the right objects without decoding the rest. Each module's self-test (self-retrieval, nearest-clip, type-and-time query) reports PASS.

THE INHERITANCE

Content-based retrieval is public domain, and it is the tool for searching any media collection by example rather than by hand-typed tags: find similar images, find similar clips, jump to the right frames of a video, detect near-duplicate media. Reduce each item to a feature signature, index the signatures, search by nearest

neighbour — the same architecture whether the signature is a simple intensity grid on a Raspberry Pi or a deep embedding in a data centre. That continuity is the real lesson: modern visual search is not a different idea from these 1990s filings, it is the same idea with a better signature, which means understanding this chapter is understanding how "search by picture" works at any scale. The machinery for searching the wordless haystack of images and video is now yours, a quarter-century after the filings that built it.

CHAPTER 32

DRAWING THE CLUSTERS (CLUSTER LAYOUT WITH CLASSICAL MDS)

THE FILING

US Patent 6,216,134 — "Method and system for visualization of clusters and classifications."

Filed 1998. Expired; public domain.

The clustering chapters found groups; this one shows them. Data often lives in many dimensions — a customer described by twenty attributes, a document by a thousand word-counts — and you cannot see twenty dimensions. This patent's claims describe placing such items on a two-dimensional map so that layout distance reflects their dissimilarity, making clusters visible to the eye. The technique is classical multidimensional scaling, and it turns a table of "how different is everything from everything" into a picture you can look at.

THE IDEA

You have a set of items and, for every pair, a measure of how different they are — a distance. It might come from many dimensions (twenty customer attributes) or from something with no coordinates at all (how dissimilar two songs sound). You want to see the structure: which items huddle together, which stand apart. But you cannot plot twenty dimensions, and a distance table is a wall of numbers. You need to place the items on a flat map so that items which are dissimilar end up far apart, and similar items end up close — preserving, as faithfully as two dimensions can, the distances you were given.

Classical multidimensional scaling (MDS) does this with a startlingly direct piece of linear algebra. Take the matrix of squared distances, "double-centre" it (a fixed arithmetic transformation that recovers, from distances alone, the geometry of the points as if you had their coordinates), and then extract its top two eigenvectors — the two directions along which the data varies most. Those two directions become the x and y of your map, and each item's coordinates are read off them. The result is the best possible flat picture of the data's distance structure: the 2-D layout whose pairwise distances match the original distances as closely as two dimensions allow. Clusters that were invisible in twenty dimensions become blobs you can point at. *patentml*'s module finds the needed eigenvectors by power iteration — repeatedly multiplying a vector

by the matrix until it aligns with the dominant direction — so the whole thing runs in pure Python with no linear-algebra library.

THE STORY

Multidimensional scaling has deep roots in psychology and statistics, where researchers from the 1950s and 60s used it to turn human judgements of similarity ("how alike are these two colours, these two sounds?") into spatial maps that revealed perceptual structure.

US6216134 brings that heritage into the database and data-mining world of the late 1990s, as a tool for visualising the clusters and classifications that mining algorithms produced — because a cluster you cannot see is hard to trust or explain. Its descendants are everywhere data is made visible: the 2-D scatter-maps that data scientists use to eyeball high-dimensional data, the visualisation front-ends of clustering tools, and the modern dimensionality-reduction methods (t-SNE, UMAP) that fill machine-learning papers with their coloured blobs — all solving the same problem this chapter names, placing high-dimensional or relational data on a plane so structure becomes visible. Every time you have seen data "projected down to 2-D" to reveal its clusters, you have seen this idea at work.

THE CODE

Double-centre the squared-distance matrix, then take its top two

eigenvectors as coordinates. Here is the core of patentml's `cluster_layout`:

```
def mds_layout(dist, seed=0):
    n = len(dist)
    D2 = [[dist[i][j]** 2 for j in range(n)] for i in range(n)]
    rmean = [sum(D2[i]) / n for i in range(n)]
    tmean = sum(rmean) / n
    B = [[-0.5 * (D2[i][j] - rmean[i] - rmean[j] + tmean) #
double-centring
        for j in range(n)] for i in range(n)]
    coords = [[0.0, 0.0] for _ in range(n)]
    for axis, (lam, vec) in enumerate(_top_eigen(B, 2, seed=seed)):
        scale = math.sqrt(max(lam, 0.0))
        for i in range(n):
            coords[i][axis] = vec[i] * scale
    return coords
```

The demo takes four points that form two clusters, given only as a distance matrix, and recovers a layout. It prints:

Cluster layout / classical MDS (from US6216134, expired):

recovered layout (up to rotation):

```
[-3.89, -0.31]
[-3.18, 0.39]
```

[3.18, -0.39]

[3.89, 0.31]

From nothing but a table of pairwise distances, MDS placed the four items on a plane — and the two clusters are unmistakable: the first two points sit together on the left (x around -3.5) together on the right (x around +3.5), exactly the two-group structure that was latent in the distances. The layout is "up to rotation" because MDS recovers relative geometry, not an absolute orientation — flip or spin the map and the distances are the same, which is all that matters for seeing the clusters. The module fuzz-tests five hundred random point sets, confirming that the layout's pairwise distances correlate above 0.98 with the originals, and reports PASS.

THE INHERITANCE

Cluster layout is public domain, and it is the tool whenever you need to see structure in something high-dimensional or defined only by similarities: visualising customer segments, laying out a similarity graph, turning "how alike is everything" into a map, giving a human a picture of what a clustering algorithm found. Classical MDS is the honest, deterministic baseline — no random seeds to fiddle, no hyperparameters, just the best flat shadow of your distance structure — and it runs in a page of pure Python. It matters most as the bridge between analysis and understanding: the clustering chapters found the groups, but this one lets a person look at them, question them,

and trust them. That ability to make data visible, and a working MDS to do it with, are yours now that this 1998 filing has expired.

CHAPTER 33

COMPARING UNLIKE SERIES (RELATED-SERIES TRANSFORMATION)

THE FILING

US Patent 6,369,819 — "Methods for visualizing transformations among related series of data."

Filed 1998. Expired; public domain.

Two data series can tell the same story at wildly different scales — a stock at £300 and a stock at £8 can rise and fall in perfect lockstep, but you would never see it if you plotted them together, because one would be a flat line at the bottom. This patent's claims describe transforming related series to a common basis so their shapes can be compared directly, and measuring how related they are. It is the small, essential toolkit of normalisation and correlation that makes unlike series comparable.

THE IDEA

The problem is that raw scale and offset hide shape. A series that runs from 100 to 120 and a series that runs from 10 to 12 may be moving identically in relative terms, but their raw numbers share no common ground, and any comparison — visual or numerical — is dominated by the difference in magnitude rather than the similarity in behaviour. To compare shape, you must first strip away scale and offset.

Two transformations do it. Z-normalisation rescales a series to zero mean and unit variation: subtract the average, divide by the spread. After this, every series lives on the same footing — its values express "how many spreads above or below its own average" — so two series with the same shape become numerically identical regardless of their original scale. Indexing to a common base does it differently for a different purpose: rebase every series so it starts at, say, 100, turning absolute values into percentage moves from a common origin (the way financial charts show several assets "rebased to 100" so their relative performance can be read off one axis). Once series are on a common basis, correlation measures how related they are: the average product of their z-normalised values, which is 1.0 when they move in perfect lockstep, 0 when unrelated, -1 when they mirror each other. Transform to a common basis to correlate — that is the whole toolkit for making unlike series speak the same language.

THE STORY

US6369819 comes from the world of data analysis and business intelligence, where analysts constantly needed to overlay and compare series that shared behaviour but not scale — sales across products of different sizes, metrics in different units, indicators that move together but live in different ranges. Its framing, "visualizing transformations among related series," points at the practical goal: transform the series so a human (or a chart) can see that they are related. The techniques themselves — z-normalisation, rebasing, correlation — are staples of statistics far older than the patent, and the filing is their industrialisation for comparing and displaying related data series in software. Their descendants are everywhere series are compared: the "rebased to 100" and "normalised" toggles on financial and analytics charts, the correlation matrices that quantify which metrics move together, and, most importantly, the preprocessing step of virtually all time-series analysis and machine learning — because z-normalisation is what lets an algorithm compare the shape of a heartbeat, a sensor trace, or a demand curve without being fooled by its amplitude. Nearly every time-series method quietly begins with the transform this chapter describes.

THE CODE

Z-normalisation puts series on a common footing; correlation measures shared shape. Here is the pair from `patentml`'s

series_transform:

```
def znorm(series):
    mean = sum(series) / len(series)
    sd = (sum((x - mean) ** 2 for x in series) / len(series)) ** 0.5
    if sd < 1e-12:
        return [0.0] * len(series)
    return [(x - mean) / sd for x in series]
```

```
def correlation(a, b):
    za, zb = znorm(a), znorm(b)
    n = min(len(za), len(zb))
    return sum(za[i] * zb[i] for i in range(n)) / n if n else 0.0
```

The demo takes two series of the same shape at very different scales and compares them. It prints:

Related-series transform (from US6369819, expired):

```
s1 z-normalized: [-1.521, -0.169, 1.183, 0.507]
s2 z-normalized: [-1.521, -0.169, 1.183, 0.507]
correlation(s1,s2): 1.0 (same shape -> ~1)
s1 indexed to 100 : [100.0, 110.0, 120.0, 115.0]
```

This is the whole idea in four lines. Series s1 (100, 110, 120, 115) and s2 (10, 11, 12, 11.5) share no raw values — one is roughly ten

times the other — yet after z-normalisation they become numerically identical, because they have the same shape. Their correlation is exactly 1.0: perfect lockstep, revealed only once scale was stripped away. And indexing `s1` to 100 shows its moves as percentages from the start (up 10%, up 20%, up 15%). Two series that looked incomparable were shown to be, in shape, the same series. The module fuzz-tests five thousand series, confirming that z-normalisation is invariant to positive scaling and shifting and that a series correlates 1.0 with its own affine copy, and reports PASS.

THE INHERITANCE

Related-series transformation is public domain, and it is the small toolkit you reach for whenever you compare series that share behaviour but not scale: overlaying metrics in different units, finding which time-series move together, rebasing for fair visual comparison, and — most of all — normalising data before feeding it to any analysis or model that cares about shape rather than magnitude. It is unglamorous and utterly foundational; z-normalisation is the quiet first step of a huge fraction of all time-series work, the thing that lets an algorithm see that two very differently-sized signals are telling the same story. Master this little transform and a whole class of "these don't seem comparable" problems dissolves. The toolkit is a dozen lines, and now that this filing has expired, it is yours.

CHAPTER 34

A LINK YOU CAN PRINT (ENCODED GRAPHIC LINKS)

THE FILING

US Patent 6,408,331 — "Computer linking methods using encoded graphics."

Filed 1999. Expired; public domain.

The book ends where the data meets the physical world. Every link so far has lived inside a computer; this one can be printed. The patent's claims describe encoding an identifier into a compact machine-readable symbol — a code carrying a check digit so it can be printed, scanned, and resolved back to the identifier, with corruption detected. It is the "physical hyperlink," the idea behind the barcode and the QR code: a database key you can put on paper, and read back with a camera.

THE IDEA

A database key is just a number or a string that points at a record. Usually it stays inside the machine. But sometimes you want to carry that key into the physical world — print it on a product, a poster, a ticket, a parcel — so that scanning it later retrieves the record it points to. To do that you need to encode the identifier into a compact symbol that survives printing and scanning, and — crucially — you need to detect when a scan has gone wrong.

The encoding is a base conversion: represent the identifier in a compact alphabet (here, 33 symbols — digits and letters, with the confusable I and O left out so a human or a smudged scan cannot mistake them), which packs a large number into a short string. The vital addition is a check digit: a value computed from the other symbols by a fixed formula (a weighted sum), appended to the code. When the code is scanned back, the reader recomputes the check digit from the symbols it saw and compares it to the one on the code. If they disagree — because a symbol was misread, smudged, or damaged — the reader rejects the code rather than resolving it to the wrong record. That single guard is what makes physical codes trustworthy: a corrupted scan fails loudly instead of silently pointing at the wrong thing. Encode compactly, carry a check digit, verify on read — the recipe behind every barcode on every product you have ever bought.

THE STORY

US6408331 sits at the turn of the millennium, when the idea of linking physical objects to digital information was becoming practical and valuable — encode a link into a printable graphic so a scan takes you to the associated record or web page. It is a database-world expression of a lineage that runs from the barcode (invented in the 1950s, ubiquitous by the 1970s, the check-digit-protected UPC on every grocery item) through to the QR code (created by Denso Wave in 1994, now on every poster, menu, and payment screen) and modern digital watermarks. All of them share this chapter's core: an identifier encoded into a compact, error-detecting symbol that bridges paper and database. The descendants are woven through daily life — the barcode at the till, the QR code you scan to see a menu or pay a bill, the tracking number on a parcel, the boarding pass on your phone — each one a printable key with a check digit, resolving to a record in some database. It is the humble, universal handshake between the physical and the digital.

THE CODE

Encode the identifier in a compact alphabet and append a check digit; on decode, verify it before trusting the result. Here is the pair from `patentml`'s `encoded_graphic_link`:

```
def encode(identifier):
    n, digits = identifier, []
    while n > 0:
        digits.append(n % _BASE); n //= _BASE    # base conversion
    digits.reverse()
    check = sum((i + 1) * d for i, d in enumerate(digits)) % _BASE    #
check digit
    digits.append(check)
    return "".join(_ALPHABET[d] for d in digits)

def decode(code):
    digits = [_ALPHABET.index(c) for c in code.upper()]
    body, check = digits[:-1], digits[-1]
    if sum((i + 1) * d for i, d in enumerate(body)) % _BASE != check:
        return None                # corruption detected -> reject
    ...
```

The demo encodes several identifiers, round-trips them, and shows a corrupted scan being rejected. It prints:

Encoded-graphic link (from US6408331, expired):

```
id    0 -> code    00 -> decode 0
id    42 -> code   18H -> decode 42
id   1000 -> code   VEP -> decode 1000
id  8675309 -> code 6GQL5B -> decode 8675309
```

corrupted check digit -> decode: None (rejected)

The encodings show the compaction: the seven-digit id 8675309 becomes the six-character code 6GQL5B, and every code round-trips cleanly back to its identifier. The last line is the whole point of the check digit — a code with one symbol corrupted decodes to None, rejected rather than resolved to some wrong record. That is the difference between a scanning system you can trust and one that silently sends a parcel to the wrong address. The module fuzz-tests twenty thousand round-trips and thousands of single-symbol corruptions, confirming perfect round-trips and that corruption is caught, and reports PASS.

THE INHERITANCE

The encoded graphic link is public domain, and it is the tool whenever you need to carry a key out of the database and into the world: label products, tickets, parcels, assets, or exhibits with a scannable code that resolves to their record, with corruption caught rather than trusted. Encode compactly, append a check digit, verify on read — a complete, robust physical-linking scheme in a page of code, no barcode library required. It is a fitting place to end the book, because it closes the loop the whole series has been tracing: a database is a way of finding things, and here the finding reaches all the way out to a symbol printed on paper and read back by a camera. The machinery that connects the physical and the digital — the

barcode's beating heart — is now yours, twenty years after the filing that carried it into the database age.

And there the tour ends. From the balanced tree that finds one key among a billion, through the bitmaps and joins and spatial indexes, down to the allocation maps and locks that manage the bytes, and out at last to a link you can print — you have walked the machinery that makes the world's data fast, findable, and connected. Every structure in these thirty-four chapters was patented by a company that has since lost its monopoly, published in full, and handed to you here as code you can run on a five-pound board. The vault is open. Go and build.

APPENDIX — THE PATENTS OF THIS VOLUME

Every algorithm in this book is anchored to a real United States patent that has since expired into the public domain. The table below lists all forty-three, in chapter order, with the filing and expiry years verified against the patent record. Where a technique predates its patent — the B-tree, hashing, Soundex, the Morton curve, multidimensional scaling — the chapter's Story said so plainly: the patent is the datable industrial artifact that carried the idea into the systems of its era, not a claim of first invention. Nothing here reveals a still-protected method; every filing's twenty-year term has run.

Ch	Patent	Title	Filed	Expired
----	-----	-----	-----	-----
1	US5832500	Method for searching an index	1996	2016
1	US5966710	Method for searching an index	1998	2018
1	US5852820	Method for optimizing entries for searching an		

index | 1996 | 2016 |
| 2 | US5878410 | File system sort order indexes | 1996 | 2016 |
| 3 | US5918225 | SQL-based database system with improved
indexing methodology | 1997 | 2017 |
| 4 | US6021409 | Method for parsing, indexing and searching
world-wide-web pages | 1998 | 2018 |
| 5 | US5893086 | Parallel file system and method with extensible
hashing | 1997 | 2017 |
| 6 | US5884307 | Updating bitmapped indexes | 1997 | 2017 |
| 6 | US5899988 | Bitmapped indexing with high granularity locking |
1997 | 2017 |
| 7 | US6141656 | Query processing using compressed bitmaps | 1997
| 2017 |
| 8 | US6185557 | Merge join process | 1998 | 2018 |
| 9 | US6487546 | Apparatus and method for aggregate indexes | 1998
| 2018 |
| 10 | US6367034 | Using query language for event filtering and
aggregation | 1998 | 2018 |
| 11 | US5987457 | Query refinement method for searching
documents | 1997 | 2017 |
| 12 | US6480835 | Method and system for searching on integrated
metadata | 1998 | 2018 |
| 13 | US6795820 | Metasearch technique that ranks documents
obtained from multiple collections | 2001 | 2021 |
| 14 | US5884303 | Parallel searching technique | 1997 | 2017 |
| 15 | US6026398 | System and methods for searching and matching

databases | 1997 | 2017 |
| 16 | US5875445 | Performance-related estimation using
pseudo-ranked trees | 1997 | 2017 |
| 17 | US5873081 | Document filtering via directed acyclic graphs |
1997 | 2017 |
| 18 | US6408270 | Phonetic sorting and searching | 1998 | 2018 |
| 19 | US6363381 | Compressed document matching | 1998 | 2018 |
| 20 | US6003036 | Interval-partitioning method for multidimensional
data | 1998 | 2018 |
| 21 | US5710915 | Method for accelerating access to a database by
clustered partitioning | 1995 | 2015 |
| 22 | US5781906 | System and method for construction of a data
structure for indexing multidimensional objects | 1996 | 2016 |
| 23 | US5839088 | Geographic location referencing system and
method | 1996 | 2016 |
| 23 | US6047236 | Geographic location referencing system and
method | 1998 | 2018 |
| 23 | US6202023 | Internet based geographic location referencing
system and method | 1999 | 2019 |
| 24 | US6148289 | System and method for geographically organizing
and classifying businesses on the world-wide web | 1997 | 2017 |
| 25 | US5960446 | Parallel file system and method with allocation
map | 1997 | 2017 |
| 26 | US6292880 | Alias-free content-indexed object cache | 1998 |
2018 |
| 27 | US6825848 | Synchronized two-level graphics processing

cache | 1999 | 2019 |
| 28 | US5828876 | File system for a clustered processing system |
1996 | 2016 |
| 29 | US5796395 | System for publishing and searching interests of
individuals | 1997 | 2017 |
| 30 | US7072846 | Clusters for rapid artist-audience matching | 2000
| 2020 |
| 31 | US5539427 | Graphic indexing system | 1994 | 2014 |
| 31 | US5867150 | Graphic indexing system | 1995 | 2015 |
| 31 | US5802361 | Method and system for searching graphic images
and videos | 1994 | 2014 |
| 31 | US6631522 | Method and system for indexing, sorting, and
displaying a video database | 1998 | 2018 |
| 31 | US6389168 | Object-based parsing and indexing of compressed
video streams | 1998 | 2018 |
| 32 | US6216134 | Method and system for visualization of clusters
and classifications | 1998 | 2018 |
| 33 | US6369819 | Methods for visualizing transformations among
related series of data | 1998 | 2018 |
| 34 | US6408331 | Computer linking methods using encoded
graphics | 1999 | 2019 |

CHAPTER 35

A CLOSING NOTE ON THE COMPANION CODE

Every algorithm in this book ships in the `patentml` library:

```
pip install patentml
```

Each module names its patent in its own docstring, carries a runnable demo that generates its own test data, and is verified against a brute-force reference — the same `PASS` you saw quoted at the end of each chapter's Code section. Nothing depends on anything outside Python's standard library. Import a tool and use it, or open its single file and read the whole thing in an afternoon. The book tells you why and when; the library lets you run, change, and break it. They are meant to be read together.

CHAPTER 36

WHERE THE SERIES GOES NEXT

The Lost Algorithms of Databases is Volume Three of a growing shelf. Volumes One and Two told machine learning's patent history; the Computer Vision volume told the story of the pixel. Ahead lie Cryptography and Data Hiding (build your own encryption), Compression (understand exactly why a file gets smaller), and Sound (how space and speech fit down a wire) — each drawn, like this one, from patents that expired into the public domain and were picked up and handed over as working code.

A patent is a bargain: a twenty-year monopoly in exchange for publishing exactly how an invention works. When it expires, the knowledge belongs to everyone — but almost nobody picks it up. This series picks it up. The vault is open. Go and build.