2013/10/28

Managed designs – Mauro servienti

Mauro.servienti@Manageddesigns.it

Goals:

What is raven

How to use it

How to avoid screwing up

Comparison with other technologies

Problem: orm misuse

Polyglot persistence:

* several persistence technologies for different kinds of data
* need for it is a function of how many use cases a product implements (e.g. erp)

Example: Analysis services + oltp

* Uses ad-hoc data structure fit for a very specific purpose

Driver behind sql/e-r model adoption: hardware cost

Storage cost estimation should include backup costs

Examples of document-oriented db:

* Exchange
* Active directory
* Lotus notes

RavenDB exploits a **native windows** component: name?

Relational model problem: scaling (two examples)

* Performance (essential for cloud/saas)
* High availability (e.g. different timezones, slecht network prestaties) (essential for cloud/saas)

CAP Theorem

Consitency high performance high availability, you can only achieve TWO

(e.g. e-r model use case: high read/writes on same table)

Possible solution: eventual consistency (but how fast????)

e.g. you have inserted an invoice even though you can’t see it ☺ (i.e. ui-related problem)

PROBLEM: change orm isolation level depending on query, orm does not encourage you to think about it

**Missing in RDBMS:**

Failure mode (distributed data, one machine fails, what then?)

Write on a, read from b , data must be the same because we want consistency but what about performance???

**Sharding**: (feature-based data partitioning technique)

Example: geographically distributed data (2 branches: London, Milan, each one stores their orders in a geographically close location, distributed queries are transparent to the application layer)

Advantages: paged distributed queries

Application configures query behavior in case of failure e.g. a remote system is down:

* I want an exception
* I want partial results

Failed rdbms implementations

* Xml columns: kill rdbms
* Azure Table storage (limited by underlying SQL server)

(odata schema)

Document-based db are NOT hot-swappable

(e.g. nhibernate + user chooses backend)

Persistency-ignorance is not easily achieved, true for reading modules

Attributes: eav/vertical table

SQL Server solution: cte

Problem: how to get Nhibernate to understand it? 🡺 user type + dialect extension

Document-based db: you get it for nothing

**NOSQL**

Denormalised data, downside: storage needs, synchronisation

No schema, advantages: no big up front design + painless to change data models

Data versioning:

* Rdbms: application version tightly coupled to db schema (must deploy at the same time)
  + Case study: multinational clothes retailer
  + How can I update my backend???? I am screwed! (downtime 🡺 lost sales)
* Document-based db: incremental schema update + application is responsible for saving data in the correct format (i.e. read version 1.0 save version 1.1, use metadata to track version information, avoid polluting domain-related information with infrastructure-related information (i.e. versioning) )
  + Problem: your application manages data versioning
  + **Golden rule: no two applications use same db (enforce separation of application lifecycles, a soa tenet)**
    - **How can two applications share data safely? 🡪 apis**

Many if not all nosql dbs use json as data persistence format

Problem: json typesystem is limited, two different applications may interpret same data differently.

Sacrifice consistence to performance and scalability.

Ravendb: Master goes down, his slave becomes readonly

Cassandra/redis/couchdb: key/value pairs, combine with elastic search to perform lookups

Graphdb: look at relationships between different pieces of data (difficult to do with document-based db or rdbms) example: neo4j

Document: represents atomic information

No way of joining/relating two documents using a docu-based db.

Order rows: part of aggregate root Order 🡺 save as a unit 🡺 some doc-dbs do not support transactions (mongo)

BISON: binary json

No limit to serializable type complexity (via JSON.Net)

ACID Transactions !!! (on write, on read by key, not the case for QUERIES)

Aggregation (map/reduce)

Transformations (e.g. query on a table and apply a view to resultset BEFORE it is returned to caller)

**Doc-based dbs encourage you to use cqrs:**

* Write aggregates (golden rule)
* Read viewmodels (depending on use case)

(protocol buffer)

Slide deck 02

AGPL license

Web folder is iis ready

How to run RavenDB

1. On my machine (start.cmd) (q: close and exit)(console app)
   1. esent(esent, c:\windows\system32\eseutils.exe /?)
      1. Backup
      2. Defrag
      3. Recovery
   2. –browser
2. Hosted on iis
   1. Rest api ☺
   2. Advantage:
      1. no administrative permissions needed ☺ (esent takes care of writing)
      2. Iis takes you back in case of failure
   3. Disadvantages:
      1. little time to boot/shutdown my process 🡺truncation problem
      2. Recycled application pools: changed to always running (machine.config in 2008)
      3. Solution: choose a hosting provider that does not replace application pools
      4. Solution: use two web roles(one public one not ☺ )
3. Windows service:
   1. Shutdown time: configurable
   2. http: uses windows svc

New database configuration

* path (~ means relative to iis root/ server.exe directory) (ravendatadir)
* log (ravenlogsdir)
* indexes
* best put them on 3 different disks (esent is write intensive)
* what you need to backup
  + northwind/data: esent data
  + northwind/logs: esent logs
  + indexDefinition: indexes definitions (need to back this up!)
  + index: save this if you can to save time☺
* raveserver.exe.config:
  + Port
  + Ravenddatadir, ravenlogdir: read only by server process
* IMPORTANT:
  + Esent: forward but NOT BACKWARD COMPATIBLE!!!!!!!!!!!!!!!!!!! (i.e. win7 -> win xp boooom!)
    - (solution: smuggler: bison ->json->reimport)
    - Replicate using master/slave
  + Voron: new storage model

In system db there’s a DOCUMENT ☺ under ‘System documents’ for each db.

(e.g. create new db, examine contents of system db, same structure of server.config)

Read only at server startup, in a production scenario a replication-based solution is often preferred.

NO CROSS-DATABASE query (by default of course ☺ )

DBS share same server resources

Raw url shown at the bottom of RavenDB studio application (same urls you’d see using fiddler)

Of course there’s also an embeddable client (sql compact –like, hosted in application, can run in-memory, 100% compatible with ravendb therefore can perform 100% integrations tests in memory and replicate data on embedded version)

Replication is http-based

**Android build (!!!!!!!!!!)**

(ravendb file structure)

client/

* Lightweight
* Abstractions
* Mvcintegration -🡪 shows queries on your MVC app pages your app is performing
* Async targeting pack 🡪 add async/await to .net4 project

Backup/

Performs server-side calls

Document

Primary key: unique in a database!

People/ (prefix that represents a set modeled by that document)(convention over configuration: replace slash with something else in settings)

(advantage: rest-friendly urls)

Ids are generate using a hilo algorithm

If I get a document wit a definite key🡺update

Else (undefined key or slash-terminated 🡺 generate key)

Content –length shown on studio page is real document length (except for metadata)

WHY HTTP???

Advantages:

Caching (provided by proxies)

Authority of information (ask)

Transport compression (iis + apply gzip)

**Security**

Server authentication

1. Windows authentication
2. Basic authentication (http header
3. OAuth :) (expose oauth server from ravendb)
4. Levels: Admin(agpl) (, none,get not allowed with agpl license)

Settings tab on system DB:

Windows authentication/api key

BUILTIN\user (BUILTIN 🡺 .net convention to refer to local machine)

\*

System

Localsystem/networkservice not appropriate because they do not have Active dir read permission!

OAUTH: great in a hosting scenario (Azure/Amazon )

Same major version means compatibility between client and server!

DTC (distributed transactions are ) Supported!

Lookup documents: exploit id format ☺ (should not change in time) (can search by prefix)(up to this point we’re using it like Cassandra (k/value store)

Orders/Italy/Lombardy/Milan/1

{

“ordernr”:”128”

}

Dealing with concurrency

Solution: Etag (onderdeel van metadata) (transformed datetime, problems?)

Metadata

* Dictionary
* Transmitted via http
* Sommige onderdelen ervan kan je niet wijzigen
* Raven entity name: used by ravendb studio ( ui) to show similar kinds of data

What you transmit via http has to be http-compatible (characters!)

Transactions protect metadata as well

ETag: involved in caching ☺

Example:

1st load: add etag in http request

2nd load: head http call 🡺 proxy returns cache info☺

**Relationships**

supported by using KEYS (see example)

NO ENFORCEMENT OF FOREIGN-KEY CONSTRAINTS !!!!

{

“some data”:”some value”,

users:[“Users/1”,”Users/2”] //models a relationship between docs, on the application level but RavenDB

}

Q: Why denormalising data makes sense??

A: avoid having to perform a query

Ex:

{

“some data”:”some value”,

Users:[

{

“username”:”A”

},{

“username”:”B”

}

] //models a relationship between docs, on the application level but RavenDB

}

SELECT N+1 problem komt ook voor in RavenDB!

(als je gebruik van een foreign key maakt!)

Relations (ask)

RavenDB is safe by default

Problem: production load scenarios different from development load scenarios🡺kaboom!

Solution:

* Paging: Ravendb forces you to paginate data (max 128 records)
* Self-tuning: behavior-driven automatic index refactoring
  + Problem: if behavior oscillates between two extremes then indexes get thrashed quickly and you spend most of your time rebuilding them

**Let’s code!**

Entry point :

* document store (similar to nhibernate session factory)
  + provides db connections
  + creation is costly
* document session
  + cheap to create
  + lifecycle spans business transaction

Connectionstring

Object initialize guarantees atomic object initialization (thanks to .net compiler )

(part of language specification)

OpenAsync:

Use async controller🡺 simplify iis threadpool management (mvc5)

OpenSession(“<system>”)🡺doesn’t work

OpneSession((String)null)🡺ok!

JSON.net 🡺 deserialisation => jobject 🡺raven🡺 ravenjobject

Get ravendb object 🡺loads metadata but you need an explicit api call to read them FROM MEMORY

Include example

Orders/2

{

Items:[{ProductId:’id0’},{ ProductId:’id1’}]

}

Session.Include(“items, ProductId”).Load<dynamic>(“Orders/2”)

(productid: is assumed to be a valid id)

Load products with productid in Items list of current object.

Problem:

Load<dynamic>(“Product/2”, “Product/2”, “Product/2”, “Foo”);

//4objects, derives from bug in version 1.0

//4 objects, (two references to “Product/2” cause creation of two elements that reference the SAME OBJECT)

//non-existing keys cause append of null elements

Using(var session=store.OpenSession()){

//a call to dispose is guaranteed

//transaction rolled back if commit not explicitly called ☺

//opening a session begins a transaction

}

Prefer explicit document aggregations to ad-hoc queries that aggregate them

e.g 2013/January/Total instead of 2013/January/\*

Hierarchies

1. single document that model a tree (suitable for small trees)
2. 1 node=> 1 document
   1. Read: expose single document to reader
   2. Write: access single node

Document store

1 doc store per server

Entry point to access ravenDB

Expensive to c reate

Singleton+threadsafe

Document session/async session

non recoverable (solve concurrency problem here but discard session)

non-thread-safe

unit of work (mapping between business transaction and db transaction)

provides identity map (key=> value)

Important

webapp: 1 http req : 1 session

desktop app:1 business transaction:1 session (live until saved)

Load<type> => try and use ‘type’ throw if you can’t deserialize to it

Load<dynamic>(“\*”)🡺use Raven-clr-type to deserialize to correct clr type

If json contains more info (i.e. extra attribute) than clr class then it gets truncated when saving

Worst case scenario

Class User{

Public string fullname{get;set;}

//but I want Firstname and lastname!

}

//use code to parse old document, use document listener

Json.net

Uses available ctors

[JsonConstructor]

public/private ctor()//make sure it gets called ☺

//prevents property initialisation

/prevents constructor execution

//used by Json.NET if no parameterless constructor found!

var obj = FormatterServices.GetUninitializedObject(typeof(Product));

//how to instantiate an interface ☺

var obj2 = FormatterServices.GetUninitializedObject(typeof(IFoo));

Id clr property does not imply json needs an id property!

Session.store(“docs/id”);//assign specific id

Session.load<dynamic>(“docs/id”);//ok

No strong typing iff your app is 100% javascript-based

(no real advantage since no server-side processing is carried out)(no viewmodel necessary)

Ensure.That (Radical.codeplex.com)

Use ExpandoObject instead of DynamicObject

ExpandoObject is an IMetadataProvider ☺

RAVENDB is NOT AT ALL suited to OLAP!!! (eventually consistent makes it almost impossible to design reports)

Real-time reporting is seldom a requirement

Up-to date implies inaccuracy, conversely accuracy implies that the data is no longer up to date

Update to Sql server: transactional

Problem with keys in sql server

Solution: unique +varchar (no update just delete), no primary key for child tables

Bundle=?

Key concepts

* Docs
* Collection(fictive concept , ui)
* Attachments (document stream, can be attached to a document via metadata e.g. picture, simplifies bkp mgmt, ideal: path to picture)
* Indexes

RavenDB ACID iff read by

* Key,
* keyStartsWith
* Etag

Eventually consistent when reading

Queries (!=load) are NOT guaranteed to return the most up-to-date data!

Data ‘freshness’ depends on load conditions, its importance depends on use cases

(

e.g. webshop wizard:irrelevant

e.g. erp: relevant (e.g. data entry of invoices)

)

RAVENDB Architecture

http-rest api (only access point to Ravendb)

background tasks

* Indexing: thread
* Reducing: 2nd step in a map-reduce process
* User tasks: plugins?

2 storage:

Document store, index store

(esent/Voron)

Document Store

Index Store

* Base (eventually consistent)
* Complex queries (influenced by indexing process)
* Queries on precomputed indexes

How to customize key generation

//customise id generation

//s:key, commands:access db within current transaction

//var tag=store.Conventions.FindTypeTagName(type).ToCamelCase();

//tag+"/"+stuff+"/"+keygenerator.NextId()

CONVENTIONS ARE CLIENT-SIDE!

Q: what happens if I change a class name from A to B

A1: FindClrType (could not keep up ☹ with the speaker)

A2: store.registrelistener(Idocumentconversionlistener); //right after store.initialize just like you’d do with nhibernate!

Raven.Client.Listeners.\*//you can find event listeners related to insertion update delete etc here

QueryListener//intercept 100% queries ☺

//use case: check query expression tree for ‘Document identifier’ (in case of a multitenant application:i.e. 2 companies same database)

Idocumentconversionlistener: base read/write operation

DocumentToEntity(string key, entity (before passing it to json.net), ravenjobject (metadata, can put versioning information here)

EntityToDocument //allows you to support 2 or more different application versions LIVE

//stuff extra property into metadata, does not change existing document schema

//load from db🡺 extract extra attributes from metadata and remap them to c# properties

You can only add listeners BEFORE a call to store.initialize()!!

Bundles do not use Listeners (since they run on the server)

Listeners are called in the order they are added!

Idea: generic listener + ioc container

ctor(IcastleWindsor) to replace a listener at runtime

Chain of responsibility: to ensure listeners are called in a specific order

Best practice: Validate data at every boundary (ui, server-side(webapi),(windows service))

How:

* data annotations on a SINGLE class that is used from BEGINNING TO END OF DATA FLOW TOWARDS STORAGE! (last validation step takes place in in documentListener)
  + e.g. from EntityToDocument
  + var ctx=ValidationContext(entity)//info on what’s happening
  + Validator.ValidateObject(entity,ctx);//throws if validation fails
* Reading from storage does not require validation

Contextual validation:

Mandatory fields depend on step (e.g. customer address in webshop is not mandatory until you want to ship)

DDD: and third-party services

Anticorruption Layer (ACL in diagram)

A reads from b

Problem: A🡨B //b changes kaboom!

Solution: A<- ACL 🡨B (AntiCorruptionLayer)

RavenDB

Index can perform calculations on data! Queries CANNOT!

Unique ids

* String
* {collection}/{uid}
  + Guid
  + Hilo
  + Sequential
  + Any string

Use case: sequential order numbering

Solution1: derive from KeyGenerator (guarantees serializability)

Solution2: add to queue (implementation?)

Metadata:

* @id
* @etag
* Last-Modified: date-time translation of etag (UTC!!!!) (therefore use datetimeoffset in .NET)(add timezone to docs to be able to reconstruct time of operation (daylight savings time etc) timezoneinfo (serializable))
* Temp-Index-Store

Problem: updates can change page order

Solution: use streaming API

Freeze data I’m streaming and DO NOT RETURN any updates after the beginning of streaming.

Expression trees are cached by using callsite! (investigate)

RavenDB QUERY API exploits LUCENE!

e.g.

session.Advanced.LuceneQuery<Order>().WhereEquals("","").WhereEquals("","")//expressions are or'ed

useful for composing queries dynamically

Server-side operators

Order by score (order by boosting/fuzzy score)

Pitfalls of automatic indexes

Problem: query without specifying indexes

automatic index generation can bring your server to its knees very quickly(can be disabled ☺ )

Solution:

Specify index

Automatic indexes:

Auto as index prefix: identifies index generated as a result of a query that does NOT specify an index.

NOT PERSISTED UNLESS queried MORE THAN x times in a given timespan!

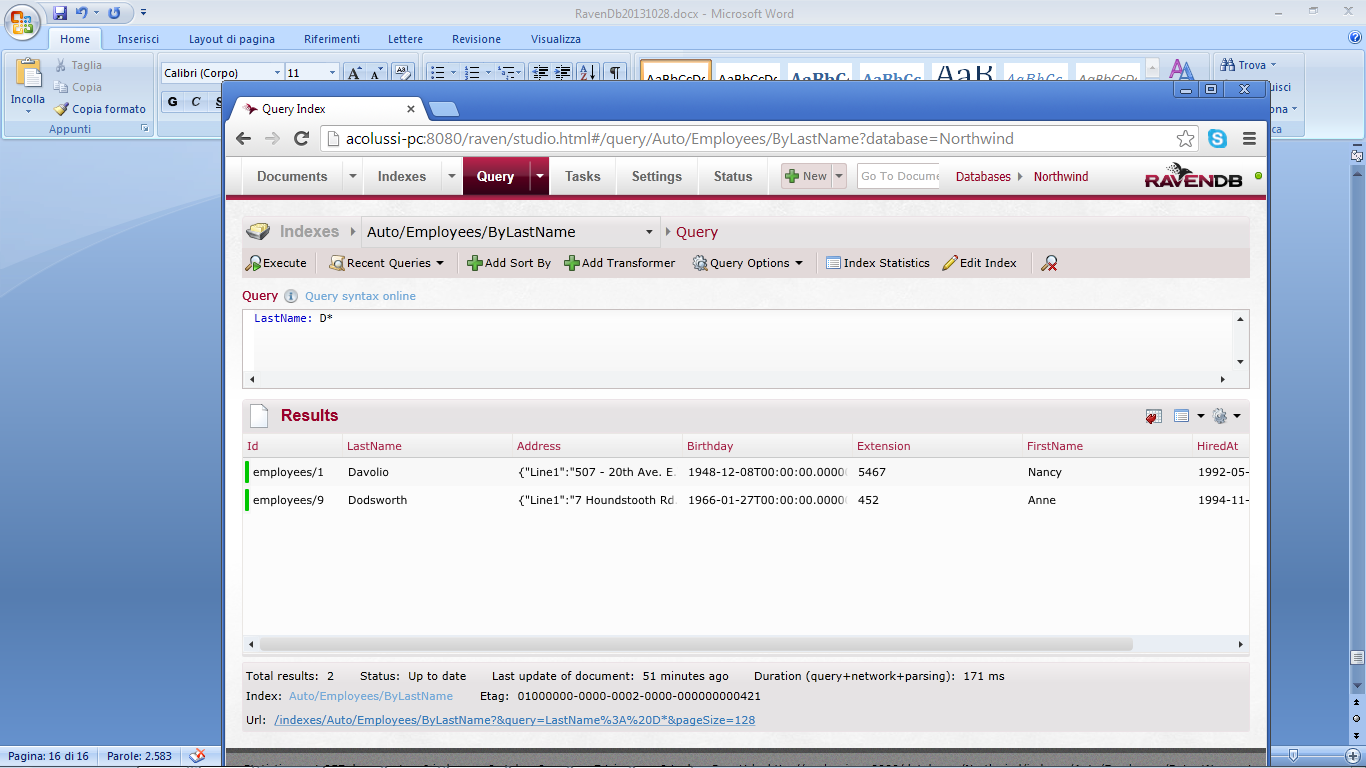
Indexes are persisted on disk as files (therefore their name is subject to file name constraints)

Index shape NOT INDEX NAME determines whether it is used.

Indexes can be locked to prevent updates from clients (

* locked (silently ignore)
* lockAndError (throws)

Example of Query on an index using RavenDb studio



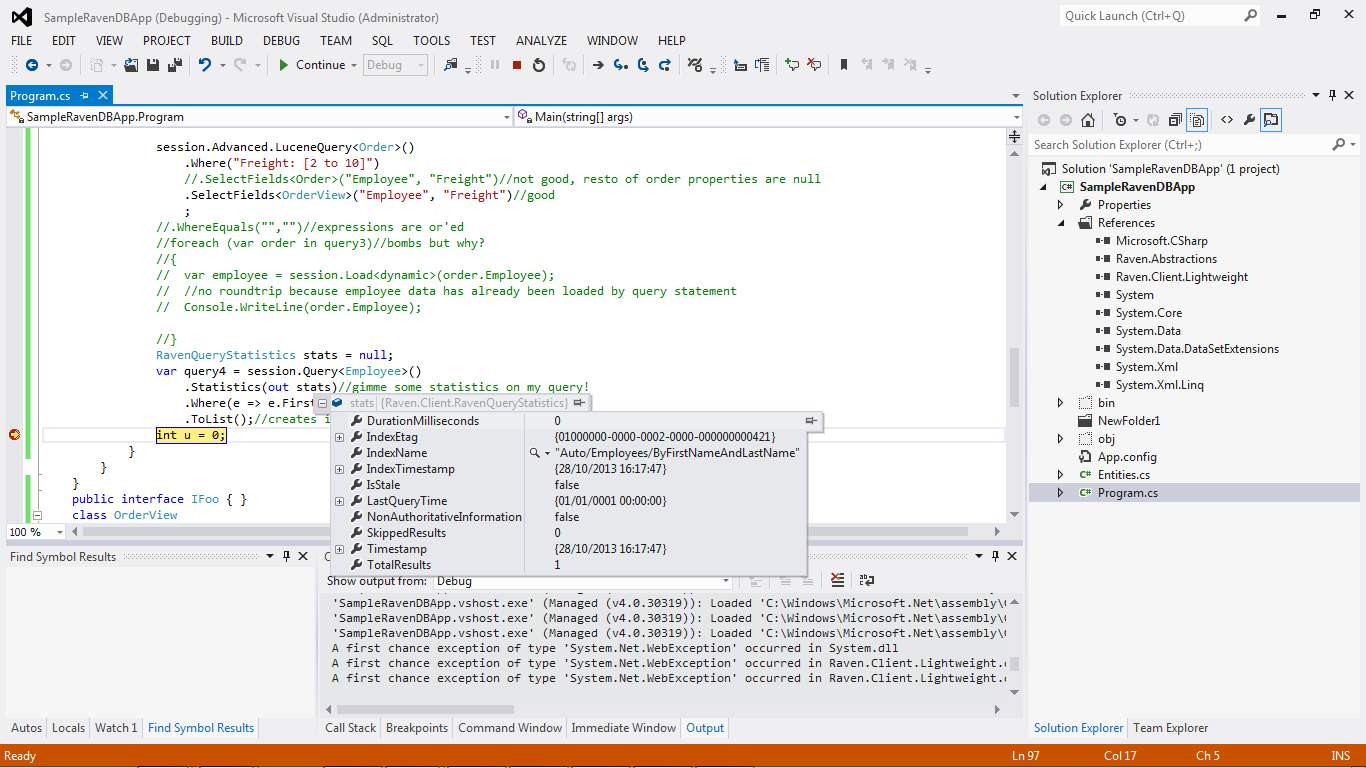
Stale Indexes:

In short: Results may not be a fully accurate picture of current data

E.g.

Indexed primary keys

Index statistics:



**Duration milliseconds**: 0 since it has used http cache!

**Index etag:**

Use case:

Bulk insert

Get metadata

Query (if etag(doc creation) < etag( from query store) then my doc is among query results so disregard stale)

**Total results:**

100% results returned (does not take any ‘linq take’ into account)

Therefore you can compute the page number with a SINGLE QUERY ☺

Lucene stores terms (column-based db, (name,key),(surname,key))

Order by on queries

Performed in DB! (unless toList is used)

NO ‘Date’ data type in Lucene!

Solution: iso format (yyyyMMdd-hh:mm:ss) if you add them to metadata and thus you DO NOT USE datetimeinfo (and .net)

Dynamic query from RavenDB Studio (use lucene format)

Autogenerated indexes are disabled if not used for a certain amount of time (Auto prefix)

After a little longer it will be marked as DELETED and therefore the very same query that created it will generate a new one!

20131029

Include// retrieve employee and orders in a single roundtrip

//read type from raven-clr-type , not found, deserialize to dynamic

//override convention, store custom types in metadata, fall back to default if no custom type is needed

What happens when you query

Returned index position

Query optimizer

Operates on Lucene queries

(no joins🡺 simple to interpret and optimize)

1. If no index specified then create index
2. Wait a few msecs
3. if index creation not completed run query anyway

Avoid create dynamic indexes in a production environment!

(it can take ages and kill your db)

Workflow

1. know your use cases
2. let ravendb create your indexes (still developing)
3. add them to your queries (c#)

Query<Class>(“Index/Name”) //not existing index🡺 exception

Not allowed: Query<Class>(“Auto/Index/Name”)

Query subclass of a hierarchy then discard documents without a certain property

Index reaction to storage changes

Save doc🡺 all indexes get stale

After a few ms 90% of them are up to date (cancel stale)

10% remain stale (rebuilding since change relevant for them)

Index orders

I have a property such as EmployeeId

Therefore the orders index has to react to changes of Employee

In next version (3.0)

If an index relies on related docs then is triggered (and choose what to do)

Else it ignores any changes

Indexing phases

1. mapping
   1. identify a subset of documents
   2. extract relevant (i.e. search attributes, such as name, phone,addr, but no email (no usecase))
   3. convert extracted information to suitable format

Use linq to let lucene know which fields are to be indexed

From doc in docs.Users

Select new

{

doc.User,

doc.Email

}

What Lucene does

|\_\_document\_id| User| Email|

|[Users/1 |Me | me@me.com |](mailto:Users/1%20|Me%20|%20me@me.com%20|)

Lucene manipulates strings and nothing else!

Tip: avoid Cartesian products

From doc in docs.Users

Select new

{

doc.User,

doc.Phones

}

What Lucene does (tokenize by splitting on spaces🡺 lucene always creates two cols: doc keys and attribute values.

|\_\_document\_id| User| Phones|

|[Users/1 |Me | phone1|](mailto:Users/1%20|Me%20|%20me@me.com%20|)

| | |phone2|

Status tab (Ravendb studio)

* Indexing errors: (induced by typos for instance e.g. post from javascript to create index)
  + Example: Create index, error, mark index as erroneous, ignore it

Server side: parse xp tree, visit it, extract fields, run query

* Statistics
  + In ram: automatic indexes are always kept in ram

INDEXES (RavenDB studio)

from doc in docs.Employees

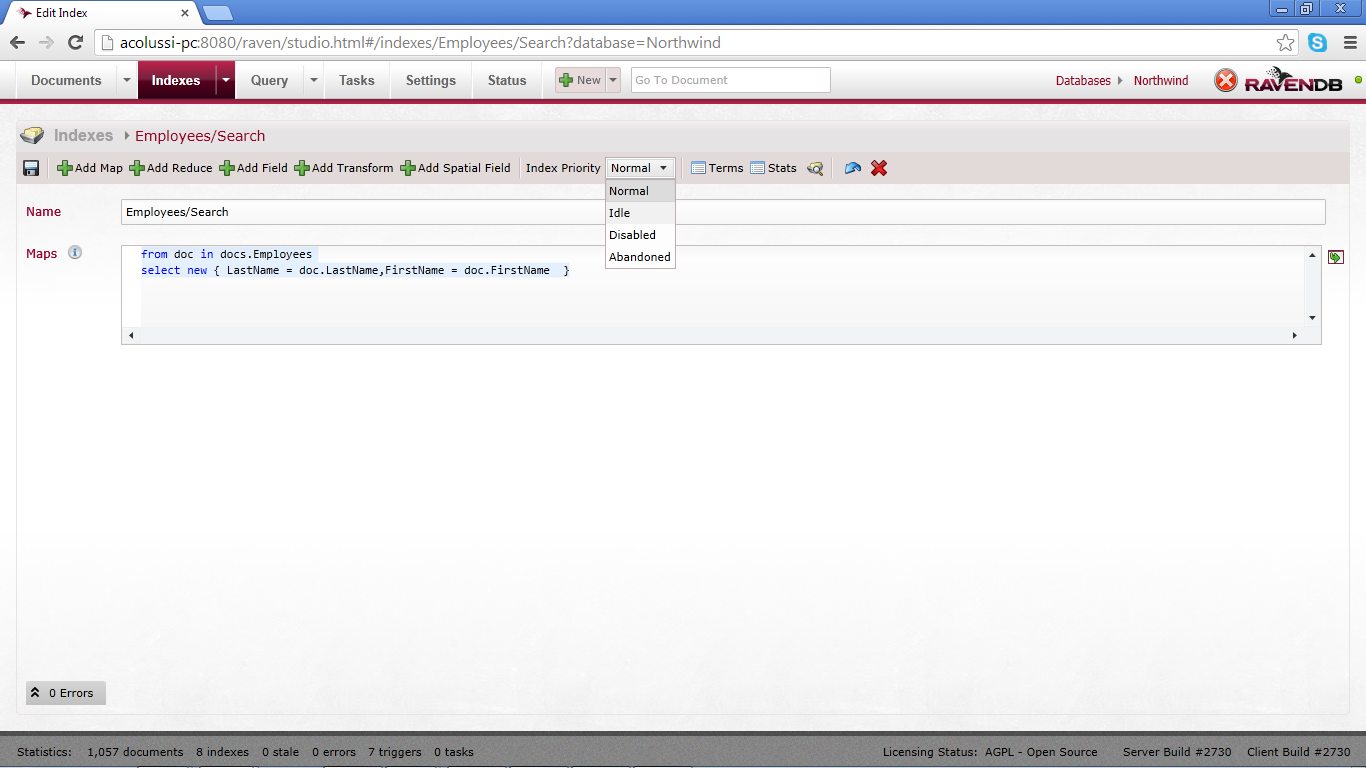
select new { LastName = doc.LastName,FirstName = doc.FirstName }

// docs.Employees is syntactic sugar to index just employee documents

//if I used docs this index would be triggered by ANY document changes

//serialization would also fail except from dynamic data type

Priority

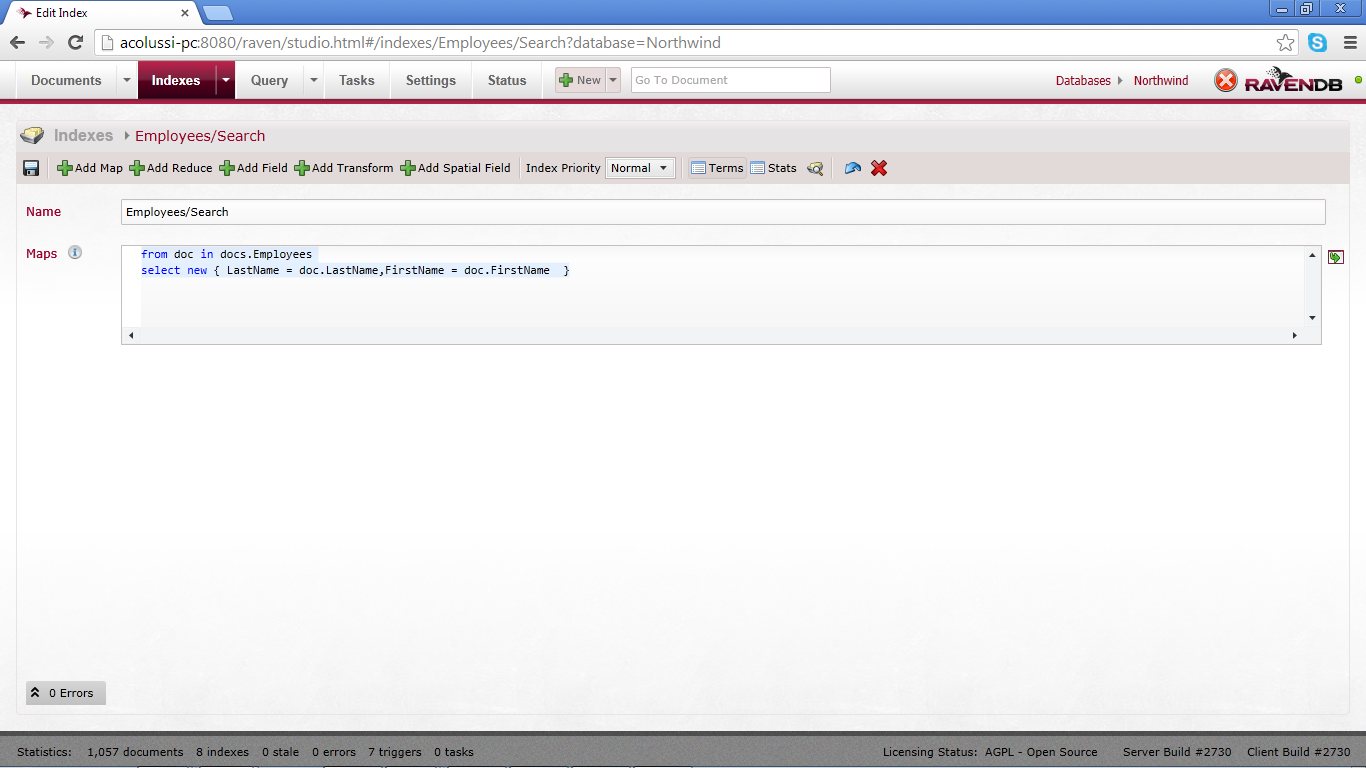


Idle: lowest priority (i.e. do not update it until you done with normal indexes) (use case: index used for reporting)

Abandoned: (autoindex not used anymore) not update anymore

Disabled: throw if used

Terms button



Lets you understand what Lucene is indexing

You can also write ‘abritrary’ C# queries and CARRY OUT COMPUTATIONS IN INDEXES!

from doc in docs.Employees

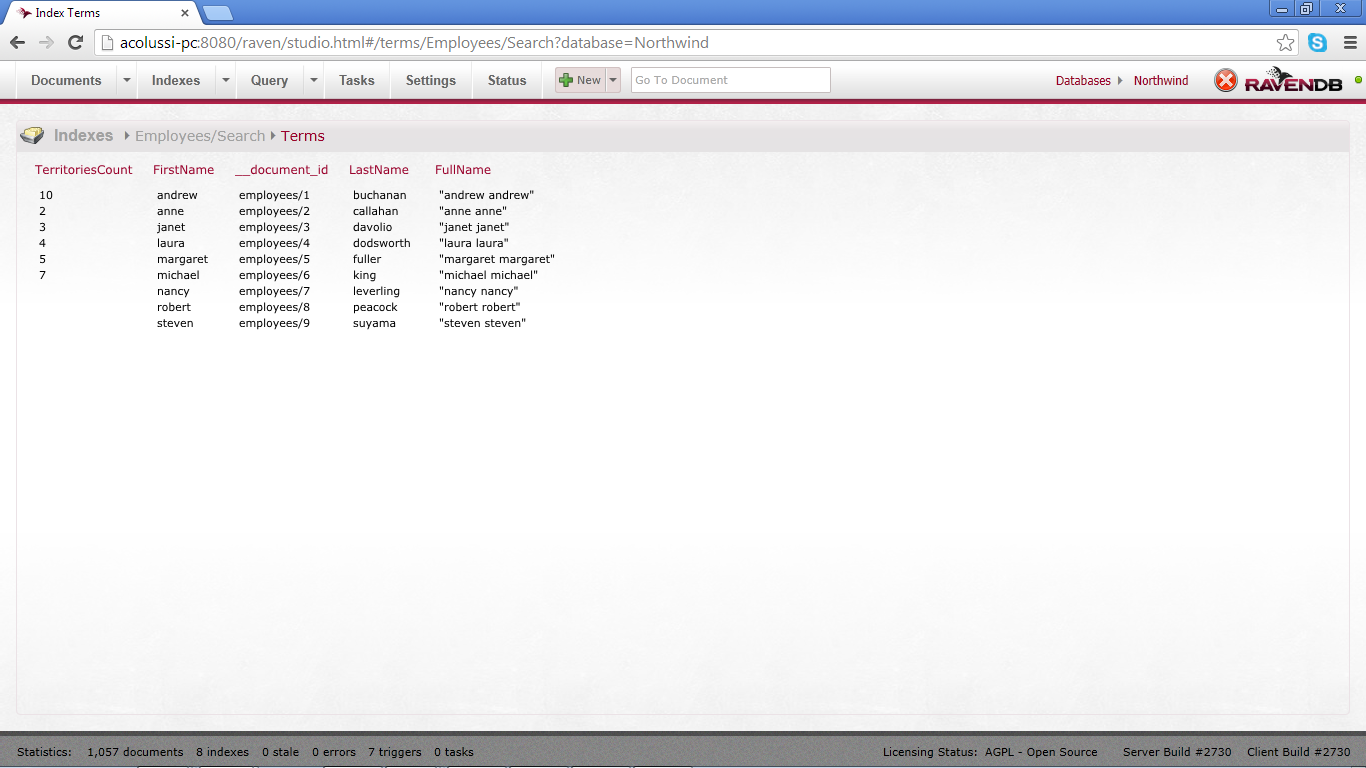
select new {

LastName = doc.LastName,

FirstName = doc.FirstName,

FullName = doc.FirstName+ " " + doc.FirstName

}



Some rows are missing TerritoriesCount 🡺 that propery is null/not existing (not true in this case, but why???)

Indexed fields CANNOT contain full stops! 🡺replace with \_ (underscore) e.g. A.b ko, A\_b ok

Summary: an index is a viewmodel!(wow-factor > infinity)

--

Problem: want to get employee data from document

from doc in docs.Orders

select new {

doc.Freight,

Employee=doc.Employee// this is the employee id! Damn!

}

Solution//join

from doc in docs.Orders

let emp= LoadDocument(doc.Employee)

select new {

doc.Freight,

Employee=emp.FirstName+ " " + emp.LastName

}

//any changes to employees render this index stale

‘Let’ is equivalent to a concatenation of Foreach statements!

Advantage of indexes: everything is server-side

Problem: ‘emp’ is a complex type, and their Annual salary amount is needed

from doc in docs.Orders

let emp= LoadDocument(doc.Employees.Select(e=>e.Id))

select new {

doc.Freight,

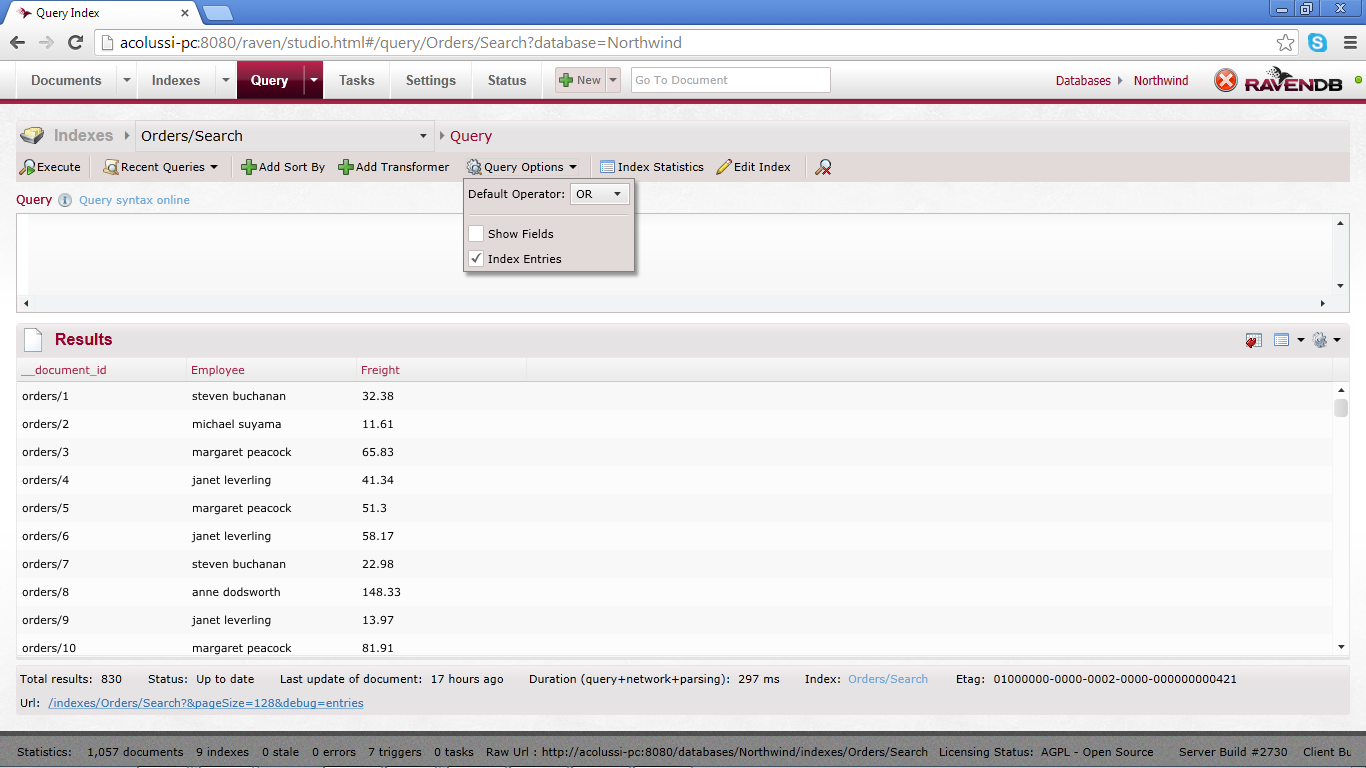
Employee=emp.SelectMany(e=>e.AnnualSalary)

}

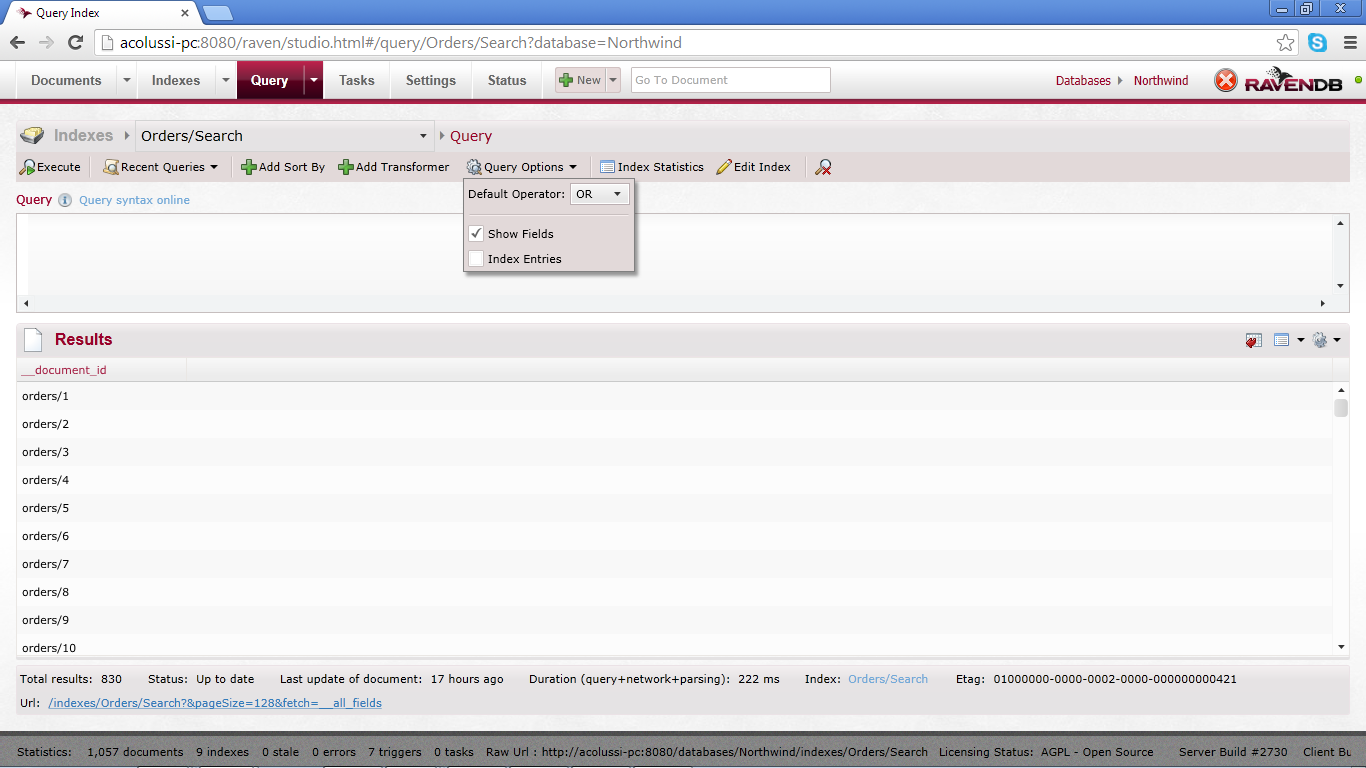
How important It is that some data is stale

In principle it is better to think of an index as a ‘ViewModel’ and sacrifice consistency to performance (if that is acceptable of course)

WaitForNonStaleResults (use only for testing)



Show persisted fields



Order\_Search.cs

Example of index creation

Orders\_Search 🡺Orders\Search

Server-side:

Ack text (linq text)

Build expression tree

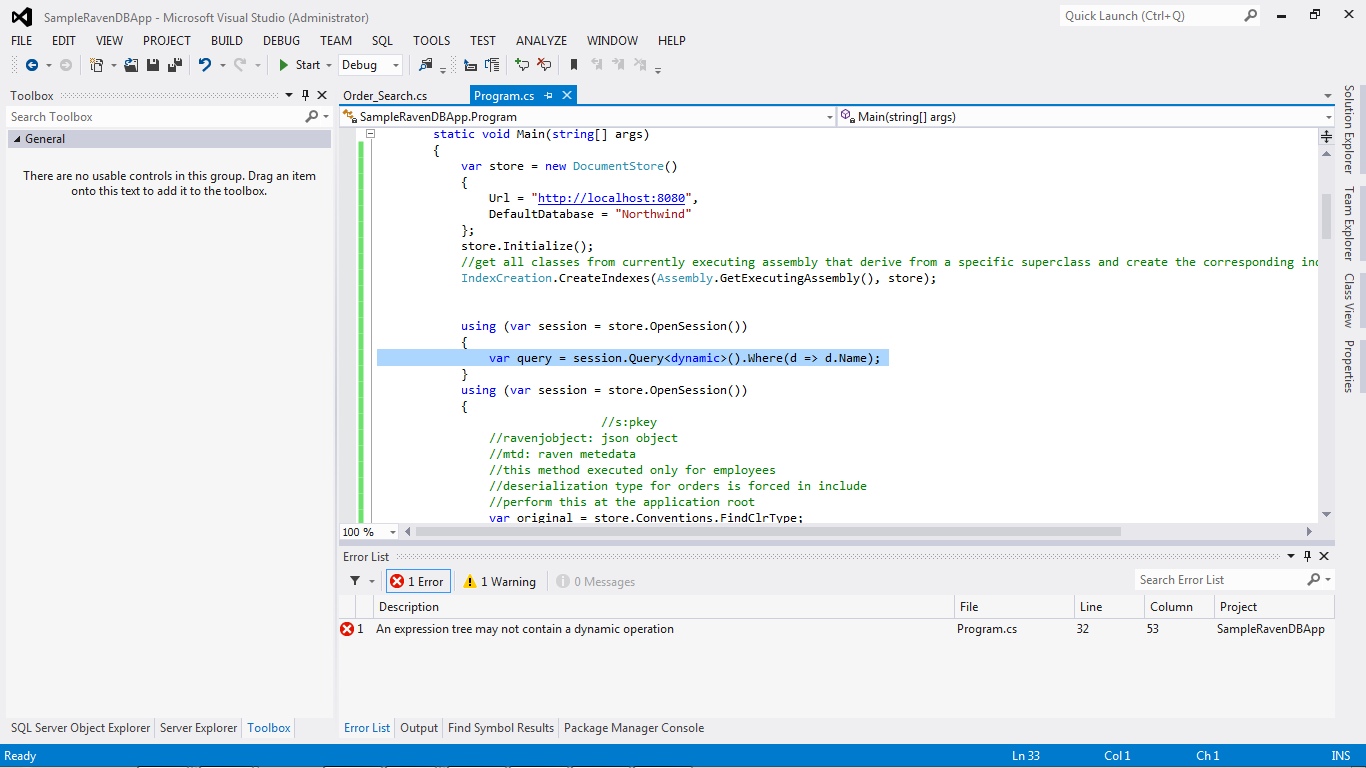
Compare expression tree

If differences change

Problem: error in index creation in code

Solution: correct on server (studio) and mark as locked ignore (prevent modifications due to app pool recycling)

Best-practice: locked ignore in a production environment



DebuggerVisualizer: shows Lucene queries (just as linq to sql does, Linq.toString() )

Problem: User wants to save a specific kind of search

Solution. Query.tostring, save to a document, replay as lucene query (no sql injection problems)

Need to specify index name explicitly!

var query = session.Query<Search\_byName.MultiMapSearch, Search\_byName>().Where(m => m.Name.StartsWith("s"))

.ProjectFromIndexFieldsInto<Search\_byName.MultiMapSearch>();

Index fields can be different from index search fields, can save you a round-trip to the document store

Index field:

Type: Analysed (can search for substring)

7ms:server side process time (in log)

Studio shows time-to-studio

Mongo:

Problem: Case-sensitive searches

Solution: store original data and lower-case data

Can cap Esent

* Memory
* Transaction memory
* Max number of docs to index