Experiences using F# for developing analysis scripts and tools over search engine query log data

Abstract. We describe our experience using the programming language F# for analysis of text query logs from the Bing search engine. The goals of the project were to develop a set of scripts for enabling ad-hoc query analysis, clustering and feature extraction as well as to provide a subset of these within a data exploration tool developed for non-programmers. Where appropriate we describe programming patterns for the text analysis domain that we used in our project. Our motivation for using F# was to explore the benefits and weaknesses of F# and functional programming in such an application environment. Our investigations showed that for the target application, common use cases involve the creation of ad-hoc pipelines of data transformations. F#'s language and library-level support for such pipelined data manipulation and lazy sequence evaluation made these aspects of the work both simple and succinct to express. We also found that when operating at the extremes of data scale, the ability of F# to natively interoperate with C# provided noticeable programmer efficiency. At these limits, reusing existing C# applications may be desirable.

1 Introduction

Query log analysis is one of the techniques that can drive improvement of the quality of search engine results. Typically, analyses of query logs require both extensive hardware resources and complex modeling techniques. Our goal in this research is to explore the suitability of the F# programming language [17] for ad-hoc analysis of query log data. We focus mainly on the support of F# for investigative programming and tool building. While the scripts and the application that we developed work on multi-core machines and out-of-main-memory scenarios, efficiency considerations were secondary to ease of use and the ability to quickly evaluate models over the data. To carry out our evaluation of F#'s suitability to the domain of query analysis we implemented a few clustering and feature selection algorithms. We incorporated the techniques that work best on our data in an application with a graphical frontend. The application allows browsing, filtering, grouping of the data and display of aggregated patterns.

The field of large scale query log data analysis is driven by the need to improve search engine quality. The dominant systems approaches in the field are MapReduce [7], SCOPE [4] and DryadLINQ [10] where the focus is on scalability. Due to the high latency caused by the large amounts of data and the sharing of cluster resources between many users, those approaches may not offer quick turnaround times. An alternative for many information retrieval researchers and practitioners is to analyze a sample of the data using interactive tools like Matlab or external commands. Those usually require that the data fits in main memory. The approach that we took with F# offers a middle ground by not requiring the data to fit in main memory while still allowing for quick compositions of primitives in scripts. Thus, the capabilities of F# for lazy evaluation over sequences that may not fit in memory, higher-order functions, succinct syntax and the possibility to define custom abstractions (e.g. pipelines of data transformations) were important for this project. We also took advantage of F# model for parallel computations. Even though we consider F# a good match to data analysis tasks such as ours, reports detailing similar experiences are rare [6]. While some of the algorithms we implemented were a good match to functional programming, we also found that some data mining algorithms that involve mutation of state did not fit the functional programming paradigm. We were still able to express them in F# using imperative programming.

The majority of open source tools in our field are written in Java. Various data processing scripts are typically written in Python or Perl. On the hand, ideas from functional programming underpin the design of MapReduce and DryadLINQ which have proven effective solutions to data analysis problem similar to ours. In our project we reap both the benefits of scripting and high-level programming abstractions that have proven useful in our field using the same programming language. We were able to build both research style scripts and seamlessly integrate them into a tool.

We describe the project in the next section. We discuss related work in Section 3. In Section 4 we detail our experience with F# and show typical use cases.

2 Project Description

Our project consisted of two stages: a) investigation of clustering algorithms applied to query logs (Table 2) and b) implementation of a graphical exploration tool (Figure 2). In the first part we implemented a few standard clustering algorithms with custom modifications. Even though clustering is well-studied topic in the statistics, data-mining and machine learning literature, finding a clustering method that works well on our dataset was a challenging task. The main reason is that most of the queries are very short. This makes it hard to estimate distances between the queries which in turn precludes off-the-shelf clustering tools to work well. Additionally, text clustering algorithms are quite sensitive to initialization and data preprocessing. While the programming language used for implementation does not guarantee success for data analysis, the use of F# did help us in the evaluation of multiple possible algorithms, parameters and data inputs. We consider that F# was a good match for this problem because of its ability to create small and easily modifiable programs. We implemented model-based K-means [21] and LDA [16]. In the second stage we incorporated the most successful clustering algorithms in a query browser application with a graphical user interface (Figure 2). We organized the application around datasets derived

Discriminative Words By Cluster
type, function, operator, pattern, types, class, cast, match, discriminated, union, interface, operators, active, matching, functions, generic, units, record, string, unit, syntax, patterns, module, int, member, measure, static, return, loop, constructor
list, array, seq, map, sample, sequence, code, monad, fold, math, fibonacci, performance, arrays, computation, recursion, tail, append, matrix, examples, cheat, comprehension, samples, sheet, immutable, monads
download, visual, studio, ctp, powerpack, 2008, 2010, express, pack, mono, power, compiler, install, 2009, split, 2, october, shell, 0, linux, documentation, beta, vs2010, 4, runtime, september, release, emacs, framework
tutorial, f, programming, async, c, language, vs, parallel, interactive, net, asp, wiki, book, tutorials, blog, center, workflows, developer, books, lazy, workflow, specification, msdn, line, command, versus, asynchronous
scale, major, minor, melodic, dorian, chords, natural, harmonic, piano, ukulele, chart, key , descending, clarinet, sax, uukulele, phrygian, ascending, pentatonic, comparison, iv, v, hash, fingering, violin, progression, min, statistical, octaves
...

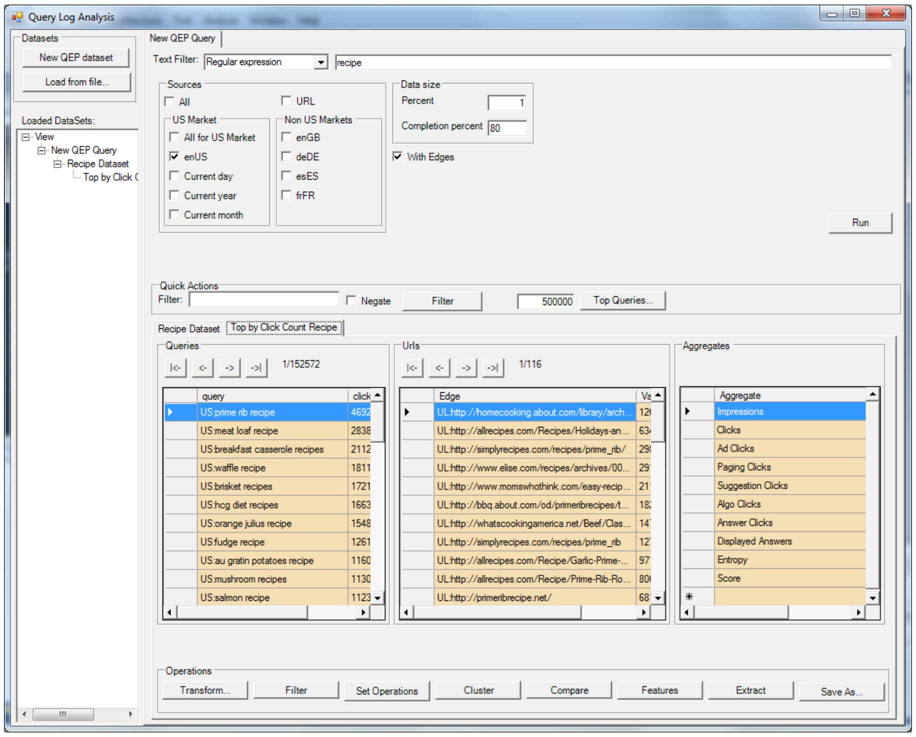
Table 1. Example of discriminative words extracted after clustering queries containing the word F#

through various operations. Given that many machine learning algorithms take a dataset and produce a new one (e.g. by filtering or weighting the data points), the selected abstraction seems natural. It is further reinforced by the F# library design which encourages that data transformations be obtained by chaining a few primitives. Thus F#'s standard library gave us a good design pattern to follow. The operations on datasets always return a new dataset without modifying an existing one in place. This approach is both for convenience, so that the user can return to an existing dataset later, and because the data may not fit in memory. We used three different kinds of datasets: 1) datasets containing queries, 2) datasets containing urls and 3) datasets containing textual features derived from the query or url datasets. We implemented the following groups of operations for query and url datasets: 1) filtering; 2) clustering; 3) selection of top data points (words, queries, urls) by various criteria; 4) grouping; 5) transformations; 6) merging; 7) extraction; 8) comparison;

All operations except extraction, comparison and feature analysis return new datasets as results. Based on insights from executing various operations the user can feed new datasets via queries from observed patterns into the system and proceed iteratively.

We carried out all computations on a high-end desktop using parallel programming techniques for the core operations. Due to the large size of the query log, multiple levels of caching take place before we can import the data in our application. We import the data from an existing distributed application written in C# via a limited number of filtering commands. We store the datasets in binary files on disk.

We demonstrate the usefulness of our tool by an example. The task of this example is to extract from the query log a list of cooking recipes. To gain an intuition about the properties of the data, the user can proceed in the following order: 1) search for queries containing the phrase "cooking recipes"; 2) explore various analysis views of top features and clusters; 3) notice that the pattern "/Recipe/X/" where X is the name of a recipe appears more than expected by chance in the urls; 4) observe that this pattern is associated with websites which have a large number of hits on cooking recipes.



Text Filter: Regular expression | recipe

Sources: [ ] All [ ] URL

US Market: [ ] All for US Market [✓ enUS] [ ] Current day [ ] Current year [ ] Current month

Non US Markets: [ ] enGB [ ] deDE [ ] esES [ ] frFR

Data size: Percent 1; Completion percent 80

[✓ With Edges]

Filter: [ ] Negate 500000

Queries 1/152572

	query	click
▶	US:prime rib recipe	4692
	US:meat loaf recipe	2838
	US:breakfast casserole recipes	2112
	US:waffle recipe	1811
	US:brisket recipes	1721
	US:hcg diet recipes	1663
	US:orange julius recipe	1548
	US:fudge recipe	1261
	US:au gratin potatoes recipe	1160
	US:mushroom recipes	1130
	US:salmon recipe	1123

Urls 1/116

	Edge	Va
▶	UL:http://homecooking.about.com/library/arch...	12
	UL:http://allrecipes.com/Recipes/Holidays-an...	63
	UL:http://simplyrecipes.com/recipes/prime_rib/	29
	UL:http://www.elise.com/recipes/archives/00...	29
	UL:http://www.momswhothink.com/easy-recip...	21
	UL:http://bbq.about.com/od/primeribrecipes/t...	18
	UL:http://whatscookingamerica.net/Beef/Clas...	14
	UL:http://simplyrecipes.com/recipes/prime_rib	12
	UL:http://allrecipes.com/Recipe/Garlic-Prime-...	97
	UL:http://allrecipes.com/Recipe/Prime-Rib-Ro...	80
	UL:http://primeribrecipe.net/	68

	Aggregate
▶	Impressions
	Clicks
	Ad Clicks
	Paging Clicks
	Suggestion Clicks
	Algo Clicks
	Answer Clicks
	Displayed Answers
	Entropy
	Score
*	

Fig. 1. Our application main screen

3 Related Work

A popular approach for large scale data processing is MapReduce[7]. MapReduce was inspired by the functions map and reduce from the Lisp programming language. MapReduce, however, is somewhat inappropriately named because it combines the functions map and reduce with a third operation: group by. A similar monolithic function was proposed in [19] before MapReduce became popular in order to solve elegantly the problem of histogram creation in functional languages. At the core of the approach of MapReduce is the use of associative operators to split the problem whose intermediate results will be aggregated. Sawzall [13] is a special purpose scripting language that is a generalization of MapReduce. The generalizations in Sawzall are in two ways: a) nested grouping and b) application of multiple associative operations within groups computed by a single pass over the data. In our application we also use associative operators (counts, random samples, filters, groups and within group statistics) and in that respect we are similar to MapReduce and Sawzall. DryadLINQ[10] is a more general approach based on a much larger number of combinators from functional programming than MapReduce. Among those operations are map, filter, group, sort, concatenate. DryadLINQ offers a distributed execution of LINQ queries by dynamic generation of query execution plans. As part of the query execution DryadLINQ also avoids out-of-memory conditions by data partitioning and serialization via reflection. While some of our code in F# uses similar combinators as Dryad we do not compute query plans but manually point intermediate results to disk when necessary. Another large scale data processing system in use is SCOPE [4]. SCOPE is modeled after SQL but allows users to plug arbitrary extractors and reducers written in any .Net language. A SCOPE program is very much like a skeleton or framework used to piece together user-defined C# functions and orchestrate the distributed execution.

Our code for file input and data transformations look similar to the Haskell code in [8]. However, the F# code is simpler because F#'s type system does not track side-effects with monads.

A basic research-style search engine in F# is described in [14]. Their implementation uses the F#'s standard library and a library for external memory algorithms. Some of our code follows a similar style. We also use a combinator-based approach similar to Haskell.Binary for specifying serializers.

F# was used for a different type of data driven research task in the TrueSkill system [6]. TrueSkill models the skill level of players in a Bayesian framework. The data experiments in TrueSkill were carried out in F#.

4 Experiences

Our experience with F# is very positive. It allowed us to immediately focus on the domain problem of exploring algorithms on query log datasets without presenting software engineering challenges. We found it beneficial to always start with the smallest program that could possibly solve the problem and then work, if necessary, towards making it more robust and efficient. We wanted to explore multiple clustering and feature selection algorithms, as well as multiple parameterizations, data inputs and initializations. F# was useful for this research problem in the following ways: 1) it helped implement well-understood baseline methods quickly and gain intuition about properties of the data, 2) allowed us to easily achieve flexible parameterization via higher-order functions and thus explore multiple inputs to the algorithms. The only bottleneck related to F# that we encountered had to do with sorting tuples of strings. We were able to work around it effectively. Other bottlenecks were related to multiple disk writes or were of algorithmic nature.

```
type Counts = Counts of int*Dictionary<string,int>
type Document = Document of (string*int)[]
let documentToWordsAndTf (Document wordsAndTf) = wordsAndTf

let countsFromStream (strm: (string*int) seq) =
    let d = strm |> Seq.countBy fst |> Dict.fromSeq
    Counts (d |> Dict.getValues |> Seq.sum, d)

let kmeans (numClusters: int) (documents: Document []): int [] =
    let numUniqueWords =
           documents
           |> Seq.collect documentToWordsAndTf
           |> Seq.distinctBy fst
           |> Seq.length
           |> float

    let negLogProb (Document wordsAndTF)  (Counts (total, counts)) =
        wordsAndTF
        |> Seq.sumBy(fun (word,tf) ->
            let wordProb = float ((Dict.getDefault 0 counts word) + 1) / ((float total) + numUniqueWords)
            - (float tf)*log wordProb)

    let docsToAssignments (clusters: (int*Counts) []) =
        let docToCluster(d: Document) = Seq.minBy (snd>>(negLogProb d)) clusters|>fst
        documents |> Array.map docToCluster

    let assignmentsToClusters (assignments: int []) =
        Array.map2 (fun clusterId doc -> (clusterId,doc)) assignments documents
        |> Seq.groupBy fst
        |> Seq.map (snd >> Seq.collect (snd>>documentToWordsAndTf) >> countsFromStream)
        |> Seq.mapi (fun i counts -> (i,counts))
        |> Array.ofSeq

    let rec loop times assignments =
        if times <= 0 then assignments
        else assignments
             |> assignmentsToClusters
             |> docsToAssignments
             |> loop (times - 1)

    let rand = System.Random 1234
    Array.init (Array.length documents) (fun _ -> rand.Next(numClusters))
    |> loop 10
```

Fig. 2. Basic implementation of model-based K-means for document clustering in F# (after [21]). This is an example of a data mining algorithm that can be expressed in functional style with the standard F# library.

4.1 Experiences with Algorithm Implementation

We found that the functional programming style fitted well the implementation of model-based K-means via standard combinators like map and group by. We show a simple implementation in figure 4 to illustrate the functional programming style we tend to use. This script gives results comparable to the ones

```
//create a parser by composing parsing combinators
let lineParser = tuple2 (parseTill "\t")
                        (restOfLine |>>= fun s -> s.Split([|'\x15'|]))
File.readLines @"input.txt"                      // read a file into a stream of lines
|> Seq.parseWith lineParser                      // parse each line using user-defined parser
|> Seq.collect (fun (query,relatedQueries) ->     // create pairs of query,relatedQuery
                       relatedQueries
                       |> Seq.map (fun relatedQuery -> (query,relatedQuery)))
|> Query.objectsToQueryNodes snd                 // use in-house library to expand the query string
                                                 // to a query object with urls
|> Seq.collect
       (fun ((query,relatedQuery), queryNode) ->
             queryNode
             |> Query.getEdges                   // extract urls from the query object
             |> Seq.map (fun (url,count) ->
                       [query; relatedQuery; url; count.ToString()] // format result
                       |> String.concat " "))
|> File.writeLines "outputFile.txt"
```

Fig. 3. An example of a typical query log processing task in F#. Similar scripts are usually used only once. Their purpose is usually to change the format or join datasets.

described in the literature for this algorithm [21]. We used this program as a starting point. We extended the program with other smoothing methods and used it for the related bisecting K-means algorithm [15]. We found that on published datasets modifications to K-means initialization improved results. Thus, the ability to easily modify programs is valuable for data processing tasks. Proponents of functional programming have often emphasized that small changes in the specification have to be reflected as small changes in the implementation. We found that this was often the case when we the used functional programming style in F#. We did an equally simple implementation in C# by imperatively updating matrix values and explicitly manipulating indexes. The resulting implementation was around 200 lines vs. 50 in F#. The high-level implementation can be easily made parallel by changing Array.map with Array.Parallel.map in the assignDocsToCluster function.

Unfortunately, we could not use the functional style for the LDA clustering algorithm. The reason is that efficient implementation of this algorithm involves mutation of state. For this algorithm we lose the benefits of functional programming, but still retain some of the benefits of F# including code conciseness resulting from the type inference system.

We found that the kinds of data mining algorithms that benefit from rapid prototyping similar to k-means include grouping, filtering and applications of statistical functions over a data collection.

4.2 Experience with Applicative Style

A large amount of the code we wrote used applicative style with F#'s function application operator as in:

```
 input |> step1 param1 |> step2 |> ... |> output
```

Run on a sample first	Run on the complete dataset
`input`	`input`
`|> sample 5000`	`// remove step |> sample 5000`
`|> step1`	`|> step1`
`|> ...`	`|> ...`
`|> output`	`|> output`

Fig. 4. An example from our practice that applicative style is easily amenable to modifications.

This code is equivalent to:

```
output(... (step2 (step1 param1 input))))
```

It is a convention to use the former style in F# code. Typically each of the steps does not use global mutable state except I/O. However, the internal implementation of most of the standard library functions and our extensions are imperative for efficiency. They may use local mutable variables as opposed to a more mathematical formulation via recursion. It is usually the case that many of the applied steps are quite simple when considered in isolation. However, when one attempts to construct directly the code which corresponds to a long pipeline, which itself may use nested pipelines, one is forced to code loops, nested loops and mix indices from different conceptual stages together. Such code quickly becomes incomprehensible and unmodifiable. In our use cases the ability to easily modify the code is highly desirable. We found this style is preferable because one can easily add and remove processing stages. A common idiom is when one wants to first run the program on a small sample of the data (figure 4.2). As can be seen in Figure 4.2 the corresponding change is very small. Had the logic been fused into a single loop such a small change would be more difficult.

A different canonical example from Information Retrieval is a document processing pipeline. In F# we use:

```
document |> tokenize |> stopword |> stem
```

Some object-oriented implementations of Information Retrieval Systems simulate this pattern by a special Pipeline interface which only applies to a stream of words. As shown in [14] and [12] this pattern does not only apply to processing words but to the index construction and query matching components of search engines.

4.3 Experience with Streams

Typical query log datasets do not fit in memory. Due to this reason the ability to stream over the data is very important. F# supports a good syntax for generating sequences and a useful library of common operations over sequences. F#'s library implements imperative streams as enumerators. There are multiple possible designs for streams each with different trade-offs. Functional streams are a more flexible alternative to imperative streams. Functional streams themselves could be implemented via recursion [2] or iteration [5]. We experimented with stream implementation via lazy lists as described in [20]. While we could create a good syntax due to F#'s workflow support, experiments revealed that lazy lists were not acceptable for our data loads. The main reason seems to be that a large number of thunks were created. While one could blame the .Net virtual machine for not optimizing such patterns, we found that even an optimizing compiler such as MLton may fail in certain complex cases (even when using the iterative implementation from [5]).

The F# workflow syntax for sequences works well when the iteration resembles a for-loop, but is harder to use in other more complex cases, for example when merging or joining two sorted sequences.

We learned to be careful when using lazy evaluation of streams and external mutable state, because the result depends on the order of evaluation. There is no protection against such problems in F# and in some cases the obtained result might be intended (e.g inserting print statements while debugging).

One should select carefully the tradeoffs between arrays and sequences. In general, one has to be careful not to unintendedly reuse a sequence twice because this might repeat long computations or produce a different result. A common example of the latter is a random number generator. Frequent materialization of sequences using arrays might end up slower than sequences because of memory accesses and cause of out-of-memory exceptions on high data loads.

F# does not have sophisticated code rewrite system for library writers to implement Haskells optimized streams. F# relies instead on the virtual machine to optimize pipelines of sequences. While sequence of pipelines (of maps, filters, etc.) are slower than direct loops, we did not find this to be a bottleneck in practice.

Programs that are expressed as a pipeline composed from various stages can be considered declarative. Therefore, the internal implementation of the combinators matters for efficiency only. The default combinators are in F#'s Seq module, but one could also use LINQ in the same way, or user-defined implementations. We have experimented with lazy functional streams, "push-based" streams, and streams based on message passing. In "push-based" streams the producer pushes items into the pipeline. The Unix command line offers a similar, but more restricted programming model, the major practical difference in our use cases being that one cannot nest other commands within a command. Stream-based interfaces are ubiquitous in functional programming.

4.4 Experience with Scripting

We give an example of a possible use of F# for scripting in our domain. In this example we read a file whose lines are formatted as "headQuery \t relatedQuery1 0x15 relatedQuery2 ...". The goal in this example is to expand each pair of query and related query to a list of corresponding urls and click counts. We use an in-house service to fill in the required data. The final result is a list of (headQuery1,relatedQuery1,url1,count1) tuples written to a file. We use the collect function twice to emit multiple values in the resulting stream. In this example, we use parsing combinators from FParsec [18] as a way to declare the input format. FParsec an implementation of the parsing combinators described in [11] for F#. Our experience is that it is possible to use parsing combinators for many ad-hoc formats which arise in our practice. We found it very useful that simple tasks such as the one described can be expressed both in short and readable code.

Custom code for input/output formatting is typical for many text processing tasks. While FParsec may be quite useful, its use can be avoided if one has control over the formatting specification. For those cases we describe the format using a simple library of serialization combinators, the most common of which are int,string, tuple2, list, etc. for parsing objects of the corresponding types. The type of the record for output can in principle be obtained using reflection, but the input type needs to be specified somehow because of the static typing in F#. We found the approach of building a schema with functions convenient. Here is some typical code:

```
// specification of the format
let schema = string @ (list string) // use of @ operator to denote a tuple of two elements
// output code
someSequence
|> Seq.outputRecords schema "filename"

// input code
Seq.inputRecords schema "filename"
|> processSequence
```

This pattern avoids the need to deal with parsing and output formatting and error prone issues such as string escaping and byte manipulation. It is also very efficient because there is no string parsing. This is a very successful example of the power of functional programming combinators because it gives gains in both programmer and program efficiency. Similar combinators are found in Haskell's Data.Binary library.

4.5 Experience with Parallel Computations

Common cases of parallel computations in our application involve filters, transformations, and groupings. Typically for each group we compute a number of statistics. In many cases the the operations we encountered were associative or could be derived from associative operations. Example of associative operations are sums, counts, random samples, filters. Textbook examples of operations that can be decomposed into associative operations are the average and the standard deviation. An important characteristic in our use cases and in many other Information Retrieval applications is that the input is usually large and read from a file. In some cases, the generated output is also large. In the field of Information Retrieval the major paradigms for grouping and reductions within each groups are MapReduce [7] and its Sawzall [13] generalization, SCOPE [4] and DryadLINQ [10]. Programs written for those systems require dedicated clusters

```
// group by key, compute the number of values and their sum for each group
// input is a sequence of (key,value) tuples

// Solution using standard F# Seq module
input
|> Seq.groupBy fst   //materializes values within groups
|> Seq.map (fun (key,values) ->
               key, (values |> Seq.length
                    ,values |> Seq.sumBy snd))
//Result is seq<'key*(int*int)>

// Solution using our "push-based" stream implementation to avoid
// unnecessary materialization of intermediate results
let spec = by (fst, split2(len(), sumBy snd))
input
|> toParallelStream
|> run spec
//Result is Dictionary<'key, (int*int)>
```

Fig. 5. A comparision between F#'s groupBy and our push-based grouping operator "by". Our implementation avoids storing intermediate results.

and specialized software. The simplest strategy we used was to split the input file into chunks with number equal to the number of CPUs, run computations for each chunk and aggregate the results.
We utilized a more complex strategy for computing groups and results within the groups. F#'s Seq module contains a groupBy function. This function, however, materializes intermediate values that fall within each group. If the input is large, usage of this function will cause the program to run out of memory. In some cases, the values within groups may not be required but only some statistics that can be computed in limited memory. We use "push-based" streams to avoid storing lists of values within groups. Instead only a few numbers are stored within each group. To produce multiple results from the same stream we use the split combinator. We can handle nested groups and support a few more combinators in addition to "by", "len", "sum" and "split". Those are collect (for emitting multiple values); map (for transformation of values); distinct (for obtaining distinct values), topBy (for obtaining top values by a criterion); values (for collecting intermediate results). We used on object-oriented implementation which allowed for chaining transformers and consumers. By using our combinators we create a specification which is passed to a "run" function. The run function takes an parallel stream, which can be split into chunks. The run function applies the specification to each chunk to produce intermediate results. When all intermediate results are computed the the run function aggregates them. Essentially, our specifications allow for composable and more general "MapReduce" style programs.

An important point is that even with a push-based stream implementation one can write programs that run out of memory. One option for us would have been to extend our combinators to handle out-of-memory conditions, but we found out that multiple disk writes are detrimental to speed-ups that can be achieved a muticore desktop. Therefore, we took care to avoid materialization

```
let emitFeatures cont = collect (fun (query,urlAndClicks) ->
                                urlAndClicks
                                |> Seq.collect (fun (url,clickCount) ->
                                     url
                                     |> featurize
                                     |> Seq.map (fun feature ->
                                                     feature,(query,url,clickCount))))
                                cont
let features = parallelStream
               |> run (emitFeatures (by(fst, len())))
let query (query,_,_) = query
let url (_,url,_) = url
let clickCount (_,_,clickCount) = clickCount
let topFeatures = features |> Dict.toSeq |> Seq.topBy 50 (snd>>desc) |> Dict.fromSeq
let spec = emitFeatures //generate features and query-url data
               (filter (fst>>Dict.hasKey topFeatures)  //filter top features
                  (by (fst, (map snd                    //take (query,url,clickCount) data
                                 (split2 (by  (query,sumBy clickCount) //compute query view
                                         ,by (url, sumBy clickCount))))))) //compute url view

parallelStream
|> run spec
```

Fig. 6. A realistic example of the use of grouping and stream splitting combinators which operate in parallel on a stream read from disk. This example also illustrates the need to split the program into two programs to avoid materialization of unnecessary results.

of large intermediate results by decomposing a program into multiple programs. The following example shows a case we encountered. In this example, the input data is in the form of a stream of queries, each query pointing to a list of urls and click counts:

```
seq{(query1, [(url11, clickCount11);(url12,clickCount12);...)]);
    (query2, [(url21, clickCount21); ...])}.
```

We would like to a) extract features from each url; b) compute top url features by number of queries generating the feature; c) compute two views for the top features: query view and url view. The features we have in mind are strings like "city=?" and are useful because they allow for extraction of values from a category. An alternative feature is the website extracted from the url. Therefore this program is useful for organizing queries by website as well. Instead of writing the program as a single expression we split the program into two phases: 1) extraction of top features; 2) given top features, compute both query and url views simultaneously (using the split combinator). In this way, we read the input data twice but do not write to disk. An alternative that is shorter to write, but slower to execute because of disk writes, would be to compute the results for each feature and its "score" in the same pass and then select best features. The solution to this task is given in figure 4.5.

While we investigated possible solutions to the simultaneous processing of split streams, we observed two styles for processing sequences: "pull-style" corresponding to F# sequences and lazy evaluation, and "push-style" corresponding to message passing and reactive programming with events. "Pull-style" can easily handle merging or joining of sequences, while it fails if a sequence needs to be split. "Push-style" fails on merging two sorted sequences. The duality between both styles has also been observed for event processing in user interfaces [1].

In addition to the combinators described above, we implemented parallel filters, histograms, transposition of a sparse matrix of (query,url) pairs in external memory and random sampling.

4.6 Experience with Tool Building

F# was also very useful for developing the user interface of the application. We used the C# user interface designer but implemented the behaviors for graphical elements from F#. Our application is parameterization rich, i.e. the user can input parameters from multiple graphical elements. To translate the user input to executable code, we mapped input from each selected user-interface element to a higher-order function. We composed all selections to obtain a function that is passed as a parameter to an operation over a dataset.
The basic design that we followed when connecting behaviors to the user interface was to use closures as callbacks. The benefit for us was that parameters such as datasets that are used in a user operation are automatically captured. This programming pattern may cause resource leaks since references to datasets are kept in closures. To solve this problem, any callback that we create returns an IDisposable object representing the assigned closure. We gather all objects corresponding to a view such as a tab and assign them to a field in the tab. Thus, when a tab is closed we release the captured resources deterministically.

4.7 Efficiency

Query mining can be quite CPU and I/O intensive depending on the task. Of the CPU intensive operations string processing and especially string sorting are the most notorious. String sorting using the generic .Net sorting functions was unacceptably slow because .Net does not implement a specialized string sorting algorithm as in [3]. An investigation of standard library implementation of sorting in Java, Haskell and Ocaml revealed lack of efficient implementation of string sorting in those libraries as well. Our implementation of string sorting was based on the reference C implementation from [3]. It was easier to translate this implementation to C# than to F#. Thus, C# can be used as a lower-level imperative language when needed. Since both C# and F# share a common representation of types no foreign-function interface is necessary.

This is in contrast to other functional languages such as OCaml or Haskell where C is the foreign-function language. When interfacing with C polymorphism cannot be easily achieved without explicitly passing a dictionary of conversion functions. Boxing and unboxing penalties are usually incurred in those cases. Thus, compared to other functional languages F# has the advantage of running on the same virtual machine as a lower-level language.

We also hit a performance issue by using comparisons of tuples. Those issues were due to F#'s new powerful equality or comparison constructs. Unfortunately, under high loads, when hashing or sorting tuples of objects we found that those constructs did not perform well. In such circumstances, in order to resolve the efficiency issue, it is best practice to switch from unnamed tuples to using named types as hash or sorting keys.

In a few cases we had to modify our code to gain efficiency but lost some readability. For example, instead of using strings we had to remap them to integer ids. To make this common task easier, we implemented a function with the following signature:

```
val withConvertToIds: ((‘a -> int) -> ‘b) -> (‘a Ids*‘b)
where
type ’a Ids =
  class
    member idToObject : int -> ’a
    member objectToId : ’a -> int
  end
```

We can run a computation within this function that remaps the ids and returns the actual result and an object which can perform the reverse map.

Another example of inefficiency is transposing a sparse matrix of queries and urls. Instead of using the obvious algorithm with Seq.collect and Seq.groupBy that works well in main memory, we had to hash manually the strings to integers and group by integer ids to avoid string sorting.

Except the issue caused by tuple comparisions, all of those pitfalls would have occurred independently of the programming language. They are either representation or algorithm dependent. During the early stages of development efficiency is not a requirement but may become later on. Due to this reason the possibility for variable mutation and C# interfacing is a strong point of F# in our use cases.

5 Conclusion

We described our experience using the programming language F# for ad-hoc analysis of query logs. Our usage of F# focused on key text analysis tasks and resulted in a graphical application for browsing logs. We focused on algorithm implementation as well as ad-hoc scripts. In some cases the algorithm could be easily expressed in functional programming style, while in others we had to resort to imperative programming. The F# language allowed us to easily formulate scripts to carry out typical tasks and we believe offered productivity gains in implementation. In our view F# is a very practical language which proved to be a good match for our usage. There are not many programming languages which can combine the conciseness of popular scripting languages like Python or Ruby with the speed of C# for typical use cases, and at the same time encourage functional programming style. Some of the programming abstractions we used have appeared in various domain specific languages for the analysis of search engine data. Those abstractions have roots in functional programming and their use is encouraging because they bring elegant and useful techniques from functional programming into mainstream practice. We believe we are the first to apply those abstractions to log analysis tasks using a language with heritage from the functional programming community.

6 Acknowledgements

We are thankful to Don Syme for his useful feedback on our paper.

References

1. Reactive extensions for .Net (Rx).
2. Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.
3. Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.
4. Ronnie Chaiken, Bob Jenkins, P. Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
5. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.
6. Pierre Dangauthier, Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill through time: Revisiting the history of chess. In *NIPS*, 2007.
7. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
8. J. R. Heard. Beautiful code, compelling evidence. functional programming for information visualization and visual analytics.
9. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
10. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.
11. Daan Leijen and Erik Meijer. Parsec: A practical parser library, 2001.
12. Marc Najork Stephen Robertson Nick Craswell, Dennis Fetterly and Emine Yilmaz. Microsoft research at trec 2009: Web and relevance feedback tracks. In *18th Text Retrieval Conference (TREC)*, 2009.
13. Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.
14. Stefan Savev. A search engine in a few lines.: yes, we can! In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 772–773, New York, NY, USA, 2009. ACM.

15. M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. Technical Report 00-034, University of Minnesota, 2000.
16. M. Steyvers and T. Griffiths. Probabilistic topic models.
17. Don Syme, Adam Granicz, and Antonio Cisternino. Expert F#, Apress, 2008.
18. Stephan Tolksdorf. Fparsec: http://www.quanttec.com/fparsec/, 2009.
19. Philip Wadler. A new array operation. In *Proc. of a workshop on Graph reduction*, pages 328–335, London, UK, 1987. Springer-Verlag.
20. Philip Wadler, Walid Taha, and David Macqueen. How to add laziness to a strict language without even being odd.
21. Shi Zhong and Joydeep Ghosh. Generative model-based document clustering: a comparative study. *Knowl. Inf. Syst.*, 8(3):374–384, 2005.