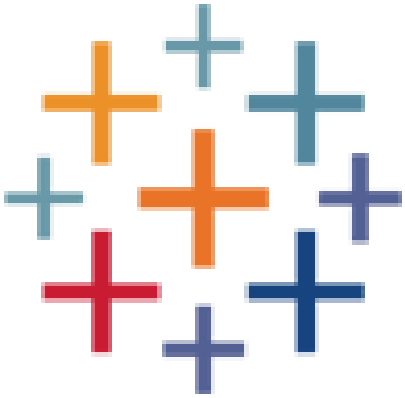

Does Anybody Really Know What Time It Is? Automating the Extraction of Date Scalars

TR #2018-01

Richard Wesley
Vidya Setlur
Dan Cory

Tableau Research
1621 North 34th Street
Seattle, WA 98103
`research.tableau.com`

Abstract

Interactive visual analytics is an effective approach for the timely analysis of data. Users who are already engaged in interactive data analysis prefer staying in the visualization environment for unavoidable data cleaning and preparation tasks to preserve their analytic flow. This has led visualization environments to include simple data preparation functions such as scalar parsing, pattern matching and categorical binning. One common scalar parsing task is extracting date and time data from string representations. Several relational database management systems (RDBMSs) include date parsing "mini-languages" to cover the wide range of possible formats, but analysis of user data from one visualization system shows that the parsing language syntax can be difficult for users to master.

In this paper, we present two algorithms for automatically deriving date formats from a column of data with minimal user disruption, one based on minimal entropy representations and another based on natural language processing techniques. Both have accuracies of over 95% on a large corpus of date columns extracted from an online data repository. One of the methods is also fast enough to produce results within the user's perceptual threshold. Moreover, we were able to avoid prohibitively expensive manual verification by using the algorithms to cross-check each other at scale.

1 Introduction

In recent years, there has been growing interest in data visualization technologies for human-assisted data analysis using systems such as Polaris [24] and Spotfire [6]. While computers can provide high-speed and high-volume data processing, humans have both the domain knowledge and the ability to process data in parallel by using their visual systems [8, 19]. Systems that rely on *both* the processing capability of computer systems and human feedback, are more effective in extracting useful knowledge from the large amounts of data being generated than either by itself.

1.1 Interactivity

Visualization systems are most effective when they are interactive, thereby allowing a user to explore data and connect it to their domain knowledge and sense of what is important without breaking cognitive flow. Exploration of data consists not only in creating visualizations, but also in creating and modifying domain-specific computations in the data model. During the analytic process, a user may discover that parts of the data are not yet suitable for analysis.

Preparing this data for analysis often requires data preparation tools external to the visual analysis environment, which forces the user to break their cognitive flow, launch another tool and reprocess her data before returning to her analysis. Recent work [21] found that the most effective systems allow users to define these calculations as part of the analytic interaction, enabling the user to stay in the flow of analysis.

We have observed that one of the most common such data preparation tasks users perform is parsing date strings into scalar date representations. Examination of author calculations in a public repository using our visualization system showed that there are about as many workbooks containing date parsing calculations (3.3%) as there are integer type conversions of any kind (3.4%). Streamlining this task is the focus of this paper.

1.2 Scalar Dates

The SQL-99 standard defines three temporal scalar types: `DATE`, `TIMESTAMP` and `TIME`, which can be further qualified as either `WITH` or `WITHOUT TIME ZONE` (the `WITHOUT` form being more common.) These types are typically implemented as fixed-point offsets from some epoch (*e.g.*, Julian Days.) This makes them compact to store using columnar compression techniques such as those in C-Store [25] and MonetDB/X100 [27], and further allows some temporal operations to be implemented efficiently using simple arithmetic. Thus from the RDBMS perspective, representing dates in scalar form provides benefits for users, both in terms of analytic richness and query performance.

From an analytic perspective, date types are dimensional (*i.e.* independent variables) and can be used as either *categorical* (simply ordered) or *quantitative* (ordered with a distance metric) fields. Categorical dates have a natural hierarchy associated with them generated by calendar *binning*, which is much easier to specify and compute with a scalar representation. Figure 1 shows an example of binned categorical dates in a year/quarter hierarchy using a bar chart.

Quantitative dates are typically used for time series on an axis that maps the underlying distance measure to display pixel distance. Figure 2 shows the same sales data in a quantitative time series rolled up to the quarter level.

Both of these examples are much easier for users to specify and manipulate when the data is modeled by scalar dates.

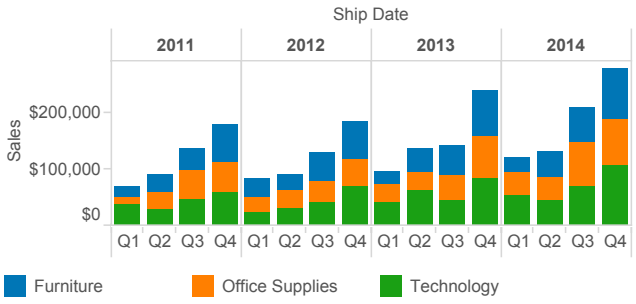

Figure 1: Categorical Date Scalars.

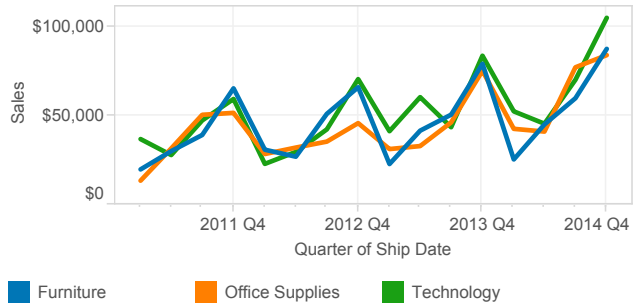

Figure 2: Quantitative Date Scalars.

1.3 Parsing Dates

A common form of the date parsing problem is converting columns of integers in the form `yyyyMMdd` to date scalars. Naïve users often solve this problem by converting the integer to a string and performing some locale-dependent string operations before casting the string back to a date, but this approach has a number of drawbacks. String operations are notoriously slow compared to scalar operations (typically 10-100x slower in modern RDBMSs). Default parsing of date formats is locale-dependent, and may not work when the parsing expression is shared across an international organization (*e.g.*, between the US and European offices). Such *ad hoc* parsing code is often hard to understand and maintain because it uses a verbose, general-purpose string-handling syntax instead of a specialized domain language.

In Section 5 we show that there are *hundreds* of distinct temporal date formats in user data sets (Figure 3). Some are common, but others can be quite idiosyncratic. Table 1 shows a selection of unusual date formats found in our corpus (the meanings of the ICU formatting codes can be found in Table 2). The first example shows a time zone in the middle of the date and a year after the time; the second shows a leading unmatched bracket and a colon between the date and time components; the third shows confusion between the seconds' decimal point and the time part delimiter; the fourth shows a two digit year apostrophe on a four digit year and the fifth shows a dash separating the date and time components. This "long tail" of formats means that these idiosyncrasies are the rule, not the exception, so a small, fixed set of formats will not produce high reliability.

ICU Format	Example
EEE MMM dd HH:mm:ss zzz yyyy	Fri Apr 01 02:09:27 EDT 2011
[dd/MMM/yyyy:HH:mm:ss	[10/Aug/2014:09:30:40
dd-MMM-yy hh.mm.ss.SSSSSS a	01-OCT-13 01.09.00.000000 PM
MM "yyyy	01 '2013
MM/dd/yyyy - HH:mm	04/09/2014 - 23:47

Table 1: Unusual Date Formats.

Several RDBMSs (*e.g.*, MySQL, Oracle and Postgres) provide row-level functions for parsing formatted dates, but a simple review of user attempts to use this functionality in published analyses showed a 15% syntax error rate when using these functions unassisted. Even with perfect syntax, the user still has to interrupt her flow to learn a formatting syntax – one that she will likely forget after this specific problem has been solved.

One approach might have been to design a graphical environment for constructing valid patterns and using visually compelling representations to reduce the cognitive load. However, this approach provides no guarantee that the result would be correct if the user misunderstood the environment. Moreover, it only solves the syntax problem – the user is still required to switch contexts.

Our solution was to develop two algorithms for automatically deriving the format string from the user's data with over 95% parsing accuracy. Both algorithms are built based on classical machine learning algorithms that learn from pattern recognition to make data-driven predictions. We developed two algorithms because we were unaware of any previous work on this problem and we needed to be able to cross-validate our results on a large test corpus. With both approaches, the user can simply specify that the column is a date, and the visualization system can respond quickly and accurately enough to avoid interrupting the user's flow.

1.4 Related Work

Data preparation has been considered an analytic bottleneck since at least the description of the Potter's Wheel system [22]. Since then, several other interactive data preparation systems have been proposed, including Data Wrangler [15] and Google Refine [14]. While effective, these systems all make assumptions about possible date formats, which we suggest are too restrictive for real world data.

Various approaches have been described for deriving regular expressions, and a good overview is provided by Li *et al.* in their paper on the ReLIE system for deriving regular expressions given a starting expression provided by a domain expert [18]. The Minimum Descriptive Length technique first described in Rissanen [23] was used in [22] to generate regular expressions.

There are several bodies of research on developing semantic parsers and grammars for interpreting time and date expressions. Lee *et al.* use a combinatory categorical grammar by combining the TIMEX3 standard [4] with contextual cues such as document creation time to determine the reference for parsing time expressions such as '2nd Friday of July' [17]. Related work by Gabor *et al.* employs a probabilistic approach for learning to interpret temporal phrases [7]. Han and Lavie developed a formalism called the Time Calculus for Natural Language (TCNL), designed to capture the meaning of temporal expressions in natural language. In this formalism, each temporal expression is converted to a formula in TCNL, which then can be processed to calculate the value of a temporal expression. Their temporal focus tracking mechanism allows correct interpretation of cases like 'I am free next week. How about Friday?', where the TCNL formula for Friday, reflects the occurrence of 'next' in the preceding sentence [12]. In all this previous research using natural language techniques, the interpretation of temporal expressions are done individually, using the presence of lexical tokens such as 'next' and 'past' along with the tense of the verb tokens.

Our work differs from these previous methods for parsing date-time expressions as we are using an entire column of data as context without the necessary presence of rich lexical identifiers to determine temporal context. This kind of data tends to be prevalent in datasets that are used in visualizations.

1.5 Contributions

- By analyzing an online corpus, we provide evidence that practical date parsing requires the ability to recognize hundreds of formats.
- We show how to extend prior work on Minimum Descriptive Length structure extraction to generate a freely available date format domain language with over 95% accuracy.
- We describe a second Natural Language Processing technique for generating the same date format domain language with similar accuracy. We describe how the basic algorithm is extended to support grammar variants and constraints unique to date formats. The parsing algorithm is extended to compute an overall dominant pattern over a data column.
- We provide evidence that the development of multiple, independent parsing algorithms provides an effective means of cross-validation on large corpora.
- We describe some limitations of this domain language that would improve its utility.

2 Preliminaries

2.1 The ICU Date Format Language

The choice of a date format syntax for generation is somewhat arbitrary, but for the purposes of exposition we will be using the formatting language defined by the ICU open-source project [5]. We chose this format because we were already using ICU in our system, we had access to the source code, and it provides localized date part data for a large number of languages (*e.g.*, month names). ICU's format syntax is typical of most syntax languages and provides a large number of date part codes, and a representative subset of it is reproduced in Table 2. The complete syntax is documented at the ICU web site [5]. Other syntaxes can either be generated directly from the algorithms, or by translating the ICU syntax into into the desired syntax. (The latter is how our visualization system internally generates formats for RDBMSs that have non-ICU syntax.)

While fairly extensive, the ICU syntax has a few limitations when working with real-world data. It has no support for 4-letter month abbreviations (*e.g.*, `Sept.`), ordinal days (*e.g.*, `July 4th`), quarter postfix notation (*e.g.*, `2Q`) and variant meridian markers (*e.g.*, `a.m.`). These limitations did not affect the results significantly, and in the future we hope to submit handlers for these formats to the ICU project.

2.2 The `DATEPARSE` Function

The date format generated by the algorithms is presumed to be a string argument to a scalar function called `DATEPARSE`, which converts a string value to a date value using the format string. Several examples of this kind of function can be found in RDBMSs such as MySQL's `STR_TO_DATE`, Oracle's `TO_TIMESTAMP` and Postgres' `TO_TIMESTAMP`. There are also programming language library implementations such as Python's `strptime` and ICU's own `DateFormat::parse`.

3 Minimal Descriptive Length

The first algorithm is a Minimum Descriptive Length [23] approach derived from the domain system presented in the Potter's Wheel system of Raman *et al.* [22]. We describe a number of extensions to

ICU Code	Interpretation
yy	year (96)
yyyy	year (1996)
QQ	quarter (02)
QQQ	quarter (Q2)
QQQQ	quarter (2nd quarter)
MM	month in year (09)
MMM	month in year (Sept)
MMMM	month in year (September)
dd	day in month (02)
EEE	day of week (Tues)
EEEE	day of week (Tuesday)
a	am/pm marker (pm)
hh	hour in am/pm 1:12 (07)
HH	hour in day 0:23 (00)
mm	minute in hour (04)
ss	second in minute (05)
S	millisecond (2)
SS	millisecond (23)
SSS	millisecond (235)
zzz	Time Zone: (PDT)
Z	Time Zone: RFC 822 (-0800)
ZZZZ	Time Zone: (GMT-08:00)
ZZZZZ	Time Zone: ISO8601 (-08:00)
’	escape for text

Table 2: ICU Format Codes.

their structure extraction system to support more complex redundancy, non-English locales, improved performance and date-specific pruning.

3.1 Domains

Potter's Wheel presents an algorithm for deriving a common structure for a set of strings by breaking each string down into a sequence of *domains*. A domain is a set of strings, with a few optional statistical properties. Each one is defined by an interface that includes:

- A required inclusion function `match` to test for membership in the domain (*e.g.*, the domain `<Digits>` returns true for `"123"` and false for `"abc"`.)
- An optional function `cardinality` to compute the number of values in the domain with a given length (*e.g.*, the domain for three letter month names (`MMM`) would return 12 for the length 3 and 0 otherwise.)
- An optional function `updateStatistics` to update statistics for the domain based on a given value (*e.g.*, the domain `MMM` might keep a histogram of how frequently each month was encountered.)
- An optional function `isRedundantAfter` to prevent consideration of a domain that is redundant. (*e.g.*, the domain `<Digits>` would be redundant after itself because there is no difference between a single run of digits and two adjacent runs.)

In our approach, we implement all of these functions, but with significant changes to `isRedundantAfter` to increase the level of pruning applied during domain enumeration.

With this interface, we can now define a set of domains for each date part that we wish to be able to parse. These are mostly straightforward enumerations and numeric ranges, each tagged with the ICU format code. Since the ICU parser is flexible about parsing single or double-digit formats, we use double-digit formats, but accept one or two digits. One important exception to this rule is for years, which are fixed width fields (2 or 4).

We found that the inclusion of arbitrary numeric domains caused the run time to grow exponentially as the number of possible matches could not be pruned intelligently. This restriction extends to domains that can contain arbitrary digit sequences (such as `<Any>`). Because of this restriction, the algorithm cannot extract non-date numeric fields.

3.2 Redundancy Extensions

A difficulty in using this kind of structure extraction is that the algorithm for enumerating structures is exponential in the number of domains. This is especially true in the date format problem because there are identical domains (*e.g.*, months and meridian hours), nearly identical domains (*e.g.*, days and hours) and there are often no field delimiters (*e.g.*, `2012Mar06134427`). To handle this, we extend the original redundancy rules with two other sets of domain identifiers:

- A set of *prunable* identifiers, which are not allowed to precede the domain. For example, once we have a month field, no other month fields should be generated. Each month domain therefore lists all the month domains in its prunable set.
- A set of *context* identifiers, one of which must have been previously generated before the domain is considered. For example, a meridian domain

can only be generated once an hour field has been found, but there may be other intervening fields.

3.3 Performance

Structure enumeration is computationally expensive, so we have added a number of enhancements to the original domain extraction algorithm to keep the run time low enough for interactivity.

3.3.1 Domain Characteristics

Date domains typically have small widths, so we found it advantageous to provide the shortest and longest match sizes for use in structure enumeration and matching. For example, months have between 1 and 2 digits, so there is no need to test matches whose length is outside this range.

Date domains are also often *uniform* in that adding more characters to a mismatch will not help. For example, a 2-digit day domain that does not match a 1-letter substring will not be able to generate a match by adding more characters.

3.3.2 Parallel Evaluation

We have also identified two opportunities for parallel computation during structure enumeration. Enumerating the matching structures for a single sample is computationally expensive, but the samples themselves are independent. Partitioning the samples allows each thread to produce a set of candidate structures, which can eventually be merged to produce a single candidate list. Once the list has been generated, the evaluation of each independent structure (computation of the MDL, recording of domain statistics and parameterizing the structure) can be conducted in parallel.

3.4 Unparameterization

Domain parameterization is an important part of generating compact representations via MDL, but it creates problems for date recognition. If (say) a set of dates contains a constant month string (*e.g.*, all values are in September) it is important to keep track of the month name domain so that the `DATEPARSE` function will parse the month. When we parameterize a constant generic `<Word>` domain, we therefore tag it with any date part domain that it matched. We then need to apply an additional pruning step to remove any structures that also found an equivalent domain (*e.g.*, two-digit month). These rules are equivalent to the context-based redundancy rules in Section 3.2, but have to be applied again after the parameterization of generics.

3.5 Global Pruning

The pruning rules used for the structure extraction reduce the search space dramatically, but they are also contextual and can only look backwards. The domains also contain a fair amount of ambiguity that requires the application of problem space knowledge. This made it necessary to add some post-generation global pruning rules:

- The set of date parts cannot contain place value gaps (*e.g.*, structures that have year and day without month are removed.)
- Similarly, the set of time parts cannot contain place value gaps and must also be in place value order (times are never written in orders such as `mhs`.)
- The existence of time parts cannot make dates incomplete (*e.g.*, patterns like year-day-hour are removed.)
- Two digit years require special handling. In particular, they cannot appear adjacent to a two-digit field if the structure contains any punctuation. (This can come up in some small early 21st century year domains where a two-digit year can masquerade as almost any numeric field *e.g.*, 08 for 2008.)

3.6 Locale

Providing an acceptable international user experience requires correct handling of the column locale. Accordingly, at the start of the structure extraction we use the locale to create a set of domains containing locale-sensitive strings such as month names. We also use the locale to map these strings to upper- and lower-case in addition to the ICU mixed case strings. This enables us to accurately compute MDL statistics without having to map the input strings at runtime (which would be slow).

Knowing the locale of a string is not always helpful. In our test data, we found numerous cases where the locale was specified (*e.g.*, Sweden) but the data was actually in English. Accordingly, we test both locales and rank the combined results.

We have built this system for the Gregorian calendar as we have little evidence of other calendars (*e.g.*, Hebrew, Islamic Civil) being used for analytics. ICU supports non-Gregorian calendars and we expect the algorithm could be extended to them as well if needed.

3.7 Ranking

MDL structure analysis produces a ranked list of format candidates, but we have found that a number of other properties of the formats should be preferred over simple compactness:

- We can apply each format to the set of samples we have, to see how well the algorithm performs. Formats with fewer parse errors are preferred.
- Date parts can be considered a place-value system, so we prefer "more significant" components (*e.g.*, month-day-year over hour-minute-second).
- If two formats from different locales give the same results, prefer the original column locale. The sample set may have missed an example where this could be important. For example, the month name for "September" is the same in both English and German, but "October" and "Oktober" are different.
- If the format has an ambiguous date order (*e.g.*, all days are less than 12), then prefer the default date order of the locale. Again, the sample set may have missed a counterexample, so this is the best option.
- Once these semantic preferences have been considered, we then prefer the more compact (MDL) representation.

The output of the algorithm is now an ordered list of formats and associated locales. These can then be used to drive a user interface that allows the user to choose between the possibilities or the top-ranking format can simply be used automatically.

4 Natural Language Processing

The second algorithm is a natural language approach to parsing date and time formats. While regular expressions can be used to match strings to known formats, we chose a grammar approach as a generalization of regular expressions. Such an approach provides for greater expressibility and less rigidity, without the expectation of the input string to *exactly* match against one or more of the known formats. We describe a particular form of grammar called a context-free grammar approach as well as extensions to the rules of the grammar that are inherent to date and time properties such as grammar variants, constraints, and assigning probabilistic weighting through a supervised learning approach.

4.1 Context-Free Grammar

ICU date formats are well defined both structurally and semantically, and can be defined by a context-free grammar (CFG). This has several advantages. First, it allows for modular syntax definitions, which simplifies grammar development and enables reuse. Second, it grants total freedom in structuring a grammar to best fit its intended use. The power of such a grammar is its expressiveness, and is not constrained by pattern rigidities that are prevalent in regular expressions and text processing [11].

A CFG consists of a set of non-terminal symbols X, a set of terminal symbols β, a start non-terminal symbol $S \in X$ from which the grammar generates the strings, and a set of production rules τ, with each rule of the form $X \to \beta$ [13].

We adopt the *Backus-Naur Form* (BNF) for defining the grammar rules for `DATEPARSE` formats. In particular, we use the *Extended Backus-Naur Form* (EBNF) as the notation is more compact and readable for frequently used constructions [11]. We employ a Cocke-Younger-Kasami (CYK) parser for syntactically parsing a column of date-time strings based on the grammar [9, 26, 16]. The parser is a dynamic programming algorithm that identifies the highest probable syntactic parse trees, given the grammar and an input date-time string.

A portion of the *EBNF* grammar is specified below:

⟨*TimeGrammar*⟩ ::= ⟨*Hours*⟩ ':' ⟨*Minutes*⟩ ':' ⟨*Seconds*⟩ ⟨*TimeZone*⟩ ⟨*AMPM*⟩ (for 12-hour formats);

⟨*DateGrammar*⟩ ::= ⟨*BigEndianDate*⟩
| ⟨*MiddleEndianDate*⟩
| ⟨*LittleEndianDate*⟩;

⟨*DateTimeGrammar*⟩ ::= ⟨*DateGrammar*⟩
| ⟨*TimeGrammar*⟩;

⟨*BigEndianDate*⟩ ::= ⟨*Year*⟩ ⟨*Month*⟩ ⟨*Day*⟩ ;

⟨*MiddleEndianDate*⟩ ::= ⟨*Month*⟩ ⟨*Day*⟩ ⟨*Year*⟩;

⟨*LittleEndianDate*⟩ ::= ⟨*Day*⟩ ⟨*Month*⟩ ⟨*Year*⟩;

⟨*Year*⟩ ::= ⟨*TwoYear*⟩ | ⟨*FourYear*⟩;

⟨*QuarterYear*⟩ := ⟨*Quarter*⟩ ⟨*Year*⟩;

⟨*Day*⟩ ::= dd (where $dd \in [01 - 31]$, depending on month/year);

⟨*Month*⟩ := ⟨*MonthWord*⟩ | ⟨*MonthNumber*⟩;

⟨*MonthWord*⟩ := '`January`' | '`February`' | '`March`' | '`April`' | '`May`' | '`June`' | '`July`' | '`August`' | '`September`' | '`October`' | '`November`' | '`December`';

⟨*MonthNumber*⟩ := dd (where $dd \in [01 - 12]$);

The parsed output is then converted to the ICU Date Format Language in order to effectively cross-validate results from the MDL algorithm. While this grammar accounts for the use of meridian markers 'a.m.' and 'p.m.' for the 12-hour format, ICU does not support these tokens, and they are simply ignored.

4.2 Grammar Variants and Constraints

A classical CFG performs poorly on inflected expressions resulting from misapplication of morphological inflection and syntactic rules. Large numbers of non-terminals are necessary for representation of all variations of such features, and the parser output would consist of many parse trees. We extend the date-time grammar by adding morpho-syntactic variants. These variants allows the parser to correct for syntactic errors (*e.g.*, usage of whitespace characters and punctuation marks), capitalization errors and abbreviations (*e.g.*, `'Mon'` for `Monday`). We use several various external corpora for such corrections [3].

The date-time grammar also includes a large number of syntactically correct but semantically invalid date-time expressions. While we have added range restrictions to symbols such as `Hour` (1–12 for 12-hour format and 1–24 for 24-hour format), `Days` (1–7), `Month` (1–12), there are special cases that need to be accounted for. For example, there is no `'November 31, 2015'`, `'February 29, 2013'`, or `'Sunday, May 5, 1965'`. November only has 30 days in any year; 2013 was not a leap year; and May 5, 1965 was a Wednesday.

While custom production rules can be added to the existing grammar to exclude such expressions, this approach is not optimal as it leads to a rather large grammar that needs to account for every single semantically valid date-time sequence of terminal symbols. Rather, we modify the existing grammar with the following additional constraints to the `Day` terminal symbol for excluding such expressions:

- **Restriction on the distribution of** 30 **and** 31: Months usually alternate between lengths of 30 and 31 days. We use x (mod 2) to get an alternating pattern of 1 and 0, and then add the constant base number of days, $\texttt{Day} = 30 + (x + 1) \pmod 2$, where $x \in [1..12]$ months. We then add a bit-masking function to the equation to correctly account for the number of days for `August` through `December` ($x \in [8..12]$): $\texttt{Day} = 30 + (x + \lfloor \frac{x}{8} \rfloor) \pmod 2$.
- **LeapYear Restriction for** `February`: While the above restriction applies to all months barring `February`, we also apply a constraint to the number of days for `February`, based on whether the year is leap year or not. For this, we define a new symbol in the grammar called `LeapYear`. If an expression containing the month `February` or any such variant (*e.g.*, `'Feb'`, `'2'`) with the day `'29'` and a year, would need to resolve the `Year` symbol to be a `LeapYear`, defined as $\texttt{Year} \pmod 4 == 0$. However, this is just an approximation. The Gregorian calendar also requires that a year evenly divisible by 100 (*e.g.*, 1800) is a leap year only if it is also evenly divisible by 400.

4.3 Probabilistic Context-Free Grammar

Pattern-recognition problems such as parsing date and time formats initiate from observations generated by some structured stochastic process. In other words, even if the initial higher-level production rule of the grammar is known (*i.e.* date, time or date-time), there could be several directions that the parser resolves to. For example, for `'5/6/2015'`, the pattern could either be `M/d/yyyy` or `d/M/yyyy`.

In the context of CFGs, probabilities have been used to define a probability distribution over a set of parse trees defined by the CFG, and are a useful method for modeling such ambiguity [10, 20]. The resulting formalism called *Probabilistic Context-Free Grammar* (PCFG), extends the CFG by assigning probabilities to the production rules of the grammar. During the process of parsing the date-time pattern, the probabilities are used to rank the pattern(s) that a given string resolves to in the grammar.

Given a CFG grammar G, the production rules are of the form $X \rightarrow \beta$ where X is the left-hand side of the rule, while β is the right-hand side. We define $\tau_G(s)$ as the set of all possible parse trees for input date-time string s. For any $X \rightarrow \beta \in \tau(G)$, the probability $p(X \rightarrow \beta) \geq 1$. In addition, $\sum_{(X \rightarrow \beta) \in \tau_G} = 1$. The parser then chooses the parse tree with the maximum probability, *i.e.* given a date-time string s, as input, we determine the highest scoring parse tree $X \rightarrow \beta$ as $max_{(X \rightarrow \beta) \in \tau(s)} p(X \rightarrow \beta)$.

Assigning initial probability weights to the grammar for computing the parse with the highest weight, tends to be a more scalable and flexible approach to disambiguation rather than building a deep knowledge ontology. We bootstrap the probability weights in the `PCFG` as follows:

- Similar to the MDL ranking properties, rules involving the non-terminal `DateGrammar` on the left-side of the rule, are assigned probability weights higher than those rules with the non-terminal being `TimeGrammar`. In practice, assigning $p(\texttt{DateGrammar} \rightarrow \beta_1) = 0.9$ and $p(\texttt{TimeGrammar} \rightarrow \beta_2) = 0.7$ leads to optimal ranking of the parse trees.
- For the non-terminal symbol `Day`, the constraint specified is a range between 01 and $28 - 31$, depending on month/year. However, for $\texttt{dd} > 12$, $p(\texttt{Day} \rightarrow dd) = 1.0$, and 0.5 otherwise. This assignment helps disambiguate days from months.

4.4 Supervised Learning

Having defined initial probabilistic weights to the `PCFG`, we employ supervised learning with a known training set of date time formats to estimate the rule probabili-

ties. The training corpus is a set of files obtained from an online data analysis corpus, described in Section 5.

The occurrence frequencies of the rules in the correct (disambiguated) parse trees can be determined from the corpus using the maximum-likelihood parameter estimates.

$$p(X \to \beta) = \frac{Count(X \to \beta)}{Count(X)} \quad (1)$$

where $Count(X \to \beta)$ is the number of times that the rule $X \to \beta$ is seen in the parse trees τ, and $Count(X)$ is the number of times the non-terminal X is seen in τ. These frequencies can then be normalized into rule probabilities. This method produces accurate probability estimates when trained on a sufficiently large corpus of disambiguated parse trees.

4.5 Context Extensions to the Grammar

The parser's success is often limited by data ambiguity, and incomplete expressions. For example an input string `'3/7/2005'` could either be interpreted as as the 7th day of March or the 3rd day of July depending on the date format used. While a single entry cannot resolve the parser's ambiguity, a column of data provides more context for determining the dominant pattern prevalent in the dataset.

The CKY parser is run over the same set of samples from each validation file to generate a probabilistic distribution of possible parse trees corresponding to a set of date-time patterns. As a second pass of the parser, starting from the most probable parse tree, each parse tree is then applied to the entire file's columnar data to determine the dominant pattern. For example, another entry in the same dataset could be `'25/3/2007'` thus increasing the probability that the format is `dd/mm/yyyy` as opposed to `mm/dd/yyyy`.

In addition to the columnar data context, the locale is also used to help maximize the probabilistic likelihood of a particular date-time pattern. The locale of the data column is used to retrieve the corresponding corpora for helping with the parsing and for correcting syntactical errors. However, similar to the locale sensitivity issue described in Section 3.6, the locale and the data may not always match. The dominant pattern is simply computed over the data column based on the ranked set of parse tree results.

5 Evaluation

For each of the two algorithms described, we want to measure the fraction of candidate date columns that the algorithm is able to recognize. We describe a large training and experimental corpus that we collected, followed by the results of applying each algorithm to that data.

5.1 Data Preparation

The training data and test data are taken from data sets published to a free, online data analysis website. This website contains data sets that users analyzed and were interested in sharing with other people through the web. It may not be representative of data in other settings such as corporate data warehouses, but each data set is one that a user was willing to invest some time in analyzing. The published data includes both the raw data and how it was used for analysis. The service limits data sets to a maximum of 100K rows stored in a columnar database with collated strings.

We collected the contents of columns with names containing any of the following strings: Date, month, created, dt (abbreviation for date), mes (month in Spanish), datum (date in German), fecha (date in Spanish), data (date in Portuguese), and the day character used in Chinese and Japanese. Roughly 95% of the data on the website is in English, but we attempted to include non-English data that was available. Fields of any data type other than date or datetime were analyzed, including strings, integers, and floats. One file containing the unique non-null values was created for each scanned column, and the column collation was stored in a second table.

Most database and spreadsheet systems already detect a limited set of date formats. For instance, typing the string `"12/31/1999"` into Microsoft Excel, is automatically interpreted as the date `1999-12-31`. The Microsoft Jet library [1] used to read these text files detects a few date formats as well. Any column already converted by Excel or Jet was not included in this study.

The data was divided into two sets, one for training and one for validation. There were 30,968 files in the training set and 31,546 files in the resulting verification set.

5.1.1 NULL Filtering

Each column was then reduced to a maximum sample set of 32. We excluded domain values that either contain the substring `NULL`, contain no digits and at most one non-whitespace character (*e.g.*, `" / / "`), or any empirically determined common `NULL` value (*e.g.*, `0000-00-00`, `NaN`).

5.1.2 Sampling

The remaining samples were hashed and sorted on the hash value, with the top 32 values retained as the column's sample. We can increase the sample size, but for dates, it appears to be an adequate number. In any case, the median number of non-null rows per domain in our test data is 50, so increasing the sample size would have little benefit.

5.1.3 Numeric Timestamps

After NULL filtering, a common class of date representation (numeric timestamps) remained that was unsuited for parsing via our date format syntax. This included Unix epoch timestamps (expressed as second, millisecond or microsecond counts from `1970-01-01`) and Microsoft Excel time-stamps (expressed as fractional days since `1900-01-01`). A simple test to check that the column was numeric and inside a specific range of values representing recent dates allowed us to tag these columns to avoid analyzing them further (and incidentally to identify them for generating a simple date extraction calculation for the user.)

5.1.4 Partial Dates

Many of the date formats that we encountered were incomplete dates, which necessitated creating rules for what date scalar they represented.

ICU's date parsing APIs allow the specification of default values for parts not found in the format. In our implementation, all time fields are set to `0` (midnight) and the date fields are set based on whether the format contains any date part specifications. When date parts are present, we use `2000-01-01` as the set of default date parts as it is the start of a leap year. When dealing with pure time formats, we model the output as a date/time and use `1899-12-30` for the date parts.

ICU will also parse Time Zones and Quarters (which we interpret as the first month of the period.) RDBMSes such as Oracle and Postgres that support time zones will be able to take advantage of identified time zone fields.

5.1.5 Training Data

Once the training data was analyzed, it was grouped by date format. A sample of each produced date format was manually labeled. This allowed us to quickly skip over very common formats like `MM/dd/yyyy` and focus our efforts on much less common formats. The samples were judged as to whether the produced format was reasonable and were tagged with correct formats if the produced format was unreasonable. A random sample of 850 columns named exactly "date", "time" or "month" (case-insensitive) were manually judged.

5.2 Minimum Descriptive Length

Testing of the MDL algorithm was performed on a 24-core Dell T7610 running Windows 7 with the data stored on a 250GB SSD.

To test the MDL algorithm, we ran it over the set of samples from each validation file to generate a ranked list of formats for the file. Each format was then applied to the entire file's data set, recording both the number of errors and the elapsed times. In cases where we generated multiple formats and the main format produced errors, we applied the second format to the unparsed strings. The summary statistics from this processing are presented in Table 3.

Number of Records	31,546
Error Rate	27.95%
Analysis Speed (μs)	2,245.04
Validation Speed (μs)	1.65
Median Not Null	50

Table 3: MDL Parsing Statistics.

The analysis speed is the average time needed per sample for structure extraction. At $2.2ms$, this is well below most human perceptual thresholds for a set of 32 samples, so any latency in command execution would be restricted to the ability of the underlying database to provide the samples for analysis in a timely manner. A columnar database (such as the one underlying our visualization system) can often supply such domains without a full table scan, further improving responsiveness.

The validation speed is the average time needed to parse a value, and provides an estimate of how fast the ICU implementation can process string values into scalars and works out to 620K values per core per second.

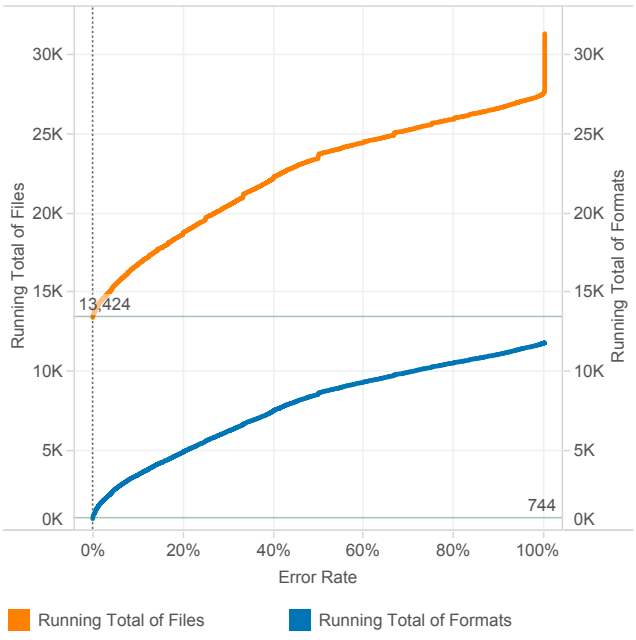

Figure 3: MDL Error Rate

The error rate reflects the fact that only about 40% of the files have an associated format that parses the

non-null values without error. To examine the error rate in greater detail, we refer to Figure 3.

The two horizontal reference lines in Figure 3 show that we found 744 distinct formats that parsed the 13,424 associated files with no error. This is a remarkable number of distinct formats and underscores the need for this kind of algorithm. Raising the error rate threshold to 5% results in about 2500 formats found in 15,000 files, or nearly half of the files in the corpus.

What do these formats look like? Figure 4 shows a histogram of the 25 most common formats containing a year format code at the 5% error threshold, color-coded by error rate. (A sample value is provided to the right of each bar for illustrative purposes.) The formats have also been filtered to files with at least 5 samples. Most of the samples are clearly dates with a wide range of formats (the format where the time zone is between the time and the year is surprisingly common.)

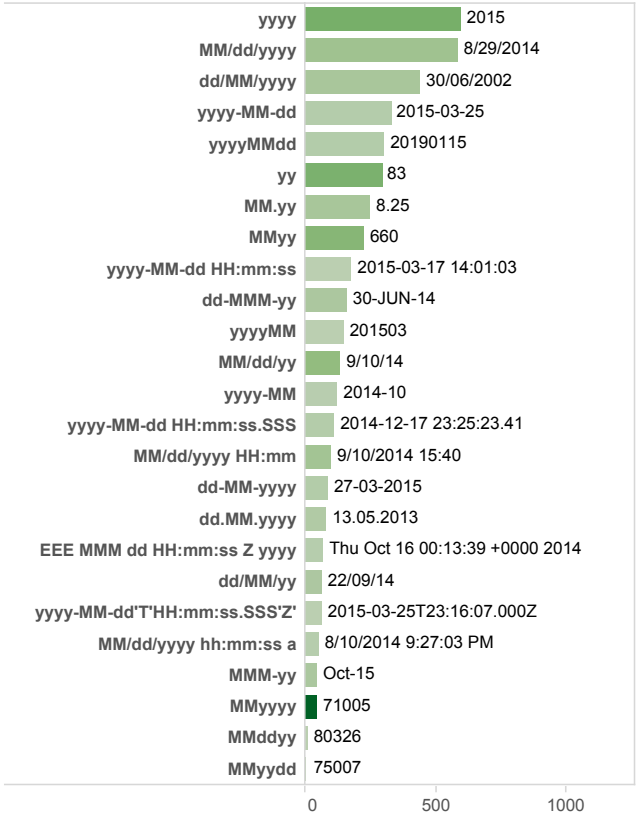

Figure 4: MDL Output

Some of the dates are clearly just numbers, but our approach is to assume that when the user tells us that the column contains dates we should find the best fit. The samples include dates from a wide range of historical sources (*e.g.*, Roman pottery dates) so we have elected to defer the semantic identification task to the user.

5.3 Natural Language Processing

We implemented the grammar, training and parsing in Python 3.4.1 using the Natural Language Toolkit libraries [2] on a quad core Dell T7600 running Windows 7. The complexity of the CYK parsing algorithm is $O(n^3|G|)$, where n is the length of the input and $|G|$ is the size of the grammar [26]. The PCFG grammar uses a set of 22 phrase-level non-terminals and 30 terminals to classify the various constituents in the corpus. Because the CYK algorithm finds the maximum likelihood solution, there are no search errors (rather probability $p = 0$, and a modification in the grammar is required to improve accuracy.

Similar to the MDL algorithm, the CYK parser is run over the same set of samples from each validation file to generate a ranked list of parse trees representing the date-time formats. As a second pass of the parser, starting from the most probable parse tree, each parse tree is then applied to the entire file's columnar data to determine the dominant pattern(s). For the second pass, we can parallelize the CYK parsing because the dependencies between reapplying the ranked list of parse trees are very sparse.

The initial average parsing speed to compute the ranked set of probable parse trees is $0.93s$, and the average time taken to compute the overall dominant pattern(s) for the entire column of data is $1.4s$. While the natural language parsing implementation is in Python as the NLTK package is easily configurable, we could expect greater speed-up in parsing performance by employing C/C++ CYK parsing libraries.

Out of the 31,546 files used for testing, the NLP parser identified a dominant format for 26,534 (84.11%) where $p >= 0.5$. 1634 unique formats were identified. Figure 5 shows a histogram of the most common formats identified containing `Year`. There are some variations compared to Figure 4 including the fact that we are not filtering the results by any error rate in the NLP parsed results.

5.4 Cross-Checking

We compared the results of the minimum description length and natural language processing algorithms. The two algorithms match for 97.9% of our validation set. The main differences in the results are due to small differences in the implementations. The minimum description length implementation recognizes formats with leading plus and minus signs, but these are almost certainly numbers, not dates. The minimum description length implementation also recognizes the Excel date formats, but this was not supported by the natural language implementation. There was one file that contained formats such as `'Fall 2000'` and `'Spring 2000'`, and the two algorithms picked different seasons in their format. One other file contained a set of integers that are not actually dates, and one algorithm

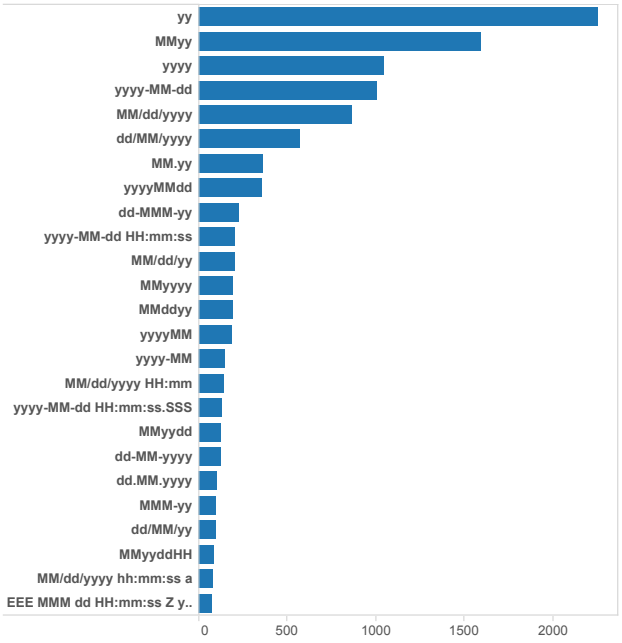

Figure 5: Most common date formats identified by the NLP algorithm.

picked `MMyyyy` format, while the other picked `HHmmss` format.

6 Future Work

During testing, we found a number of columns that we were not able to process with these techniques. Many columns contained multiple date formats. We would like to recognize this situation and generate predicate-based calculations (*e.g.*, if the string matches format 1 then apply format 1 else ...) to increase the accuracy of the results and thereby make the experience even more seamless for the user. Other columns contain date ranges, which we would like to handle by generating multiple calculations, possibly by combining regular expressions with date parsing.

In this paper, we have considered the parsing of strings, but dates are often formatted as integers (*e.g.*, `201507016`). It is significantly faster to decompose integers into date parts using arithmetic operations (*e.g.*, `mod` and `div`) than by using locale-sensitive string parsing functions. Timestamp preparation from numeric representations is a related task that we would also like to automate.

In the course of our research, we have also identified a number of date part variants (*e.g.*, ordinal dates, four-letter month abbreviations, alternate meridian markers and postfix quarter syntax) that we would like to commend to the ICU project, along with possible implementations.

7 Conclusion

In this paper, we have described two effective algorithms for extracting date format strings from a small set of samples, one using a minimum descriptive length approach and one using natural language techniques. Both algorithms are accurate enough to be used automatically without user involvement. The MDL algorithm is also fast enough to deploy in an interactive environment, freeing users from the need to interrupt their cognitive flow during analysis in order to learn a formatting language.

While validating the algorithms on a large corpus, we also found that the number of distinct formats in the wild is surprisingly high, and demonstrates the wisdom of including general-purpose date parsing functions in data visualization tools, data cleaning tools and RDBMSs. In particular, it is interesting to note that the most prominent open source RDBMSs (*e.g.* MySQL and Postgres) both have a built-in version of `DATEPARSE`, possibly reflecting that this is a common need that gets implemented when users are empowered to extend the function library of an RDBMS.

References

[1] Microsoft Odbc Desktop Database Drivers. `https://msdn.microsoft.com/en-us/library/ms711711(v=vs.85).aspx`.

[2] Natural Language Toolkit. `http://www.nltk.org/`.

[3] Natural Language Toolkit Corpora. `http://www.nltk.org/nltk_data/`.

[4] Timex3: A Formal Specification Language for Events and Temporal Expressions. `http://www.timeml.org/site/publications/timeMLdocs/timeml_1.2.1.html`.

[5] International Components for Unicode. `http://site.icu-project.org/`, July 2015.

[6] C. Ahlberg. Spotfire: An Information Exploration Environment. *SIGMOD Rec.*, 25(4):25–29, Dec. 1996.

[7] G. Angeli, C. D. Manning, and D. Jurafsky. Parsing Time: Learning to Interpret Time Expressions. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL HLT '12, pages 446–455, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[8] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to

the development of graphical methods. *Journal of the American Statistical Association*, 79(387):pp. 531–554, 1984.

[9] J. Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

[10] M. Collins. Head-Driven Statistical Models for Natural Language Parsing. *Comput. Linguist.*, 29(4):589–637, Dec. 2003.

[11] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[12] B. Han and A. Lavie. A Framework for Resolution of Time in Natural Language. 3(1):11–32, Mar. 2004.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[14] D. Huynh and S. Mazzocchi. Google Refine. `http://code.google.com/p/google-refine/`, July 2015.

[15] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3363–3372, New York, NY, USA, 2011. ACM.

[16] T. Kasami. An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[17] K. Lee, Y. Artzi, J. Dodge, and L. Zettlemoyer. Context-Dependent Semantic Parsing for Time Expressions. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 1: Long Papers*, pages 1437–1447, 2014.

[18] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular Expression Learning for Information Extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[19] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, Apr. 1986.

[20] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[21] K. Morton, R. Bunker, J. Mackinlay, R. Morton, and C. Stolte. Dynamic Workload Driven Data Integration in Tableau. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 807–816, New York, NY, USA, 2012. ACM.

[22] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 381–390, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14(5):465–471, Sept. 1978.

[24] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases. *Commun. ACM*, 51(11):75–84, Nov. 2008.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[26] D. H. Younger. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2):189–208, February 1967.

[27] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.