Fast pattern matching

Alexander Kramer

Mittwoch, 19. Juni , 2002

1 Problemstellung

Man möchte die Daten mit einem Index belegen, der es erlaubt Datensätze schneller zu finden, die eine vorgegebene Teilmenge enthalten. Ein einfaches Beispiel ist die Suche nach einem Teilstring.

2 Mögliche Lösung

Wir erstellen zuerst eine sortierte Menge der Zeichen(ketten), die wir mit der Menge der Patterns bezeichnen. Für einfaches Beispiel nehmen wir an, die Menge der Patterns enthält 4 Zeichen {'a','b','c','d'}. Dann bilden wir für jeden Datensatz ein Bitmuster, im welchen wir das Vorhandensein eines Patterns mit einem gesetzten Bit quittieren :

```
1) ab   -> 1100
2) cb   -> 0110
3) bc   -> 0110
4) abdc -> 1111
5) wx   -> 0000
```

Falls wir jetzt nach einem Teilstring "bc" suchen möchten, so erstellen wir ebenfalls ein Bitmuster dafür *(bm(search)=0110)*. Ein Datensatz kommt für weitere Suche nur dann in Frage, wenn Bitmuster des gesuchten Strings von dem Bitmuster des Datensatzes *(bm(data))* überdeckt wird. Mit *candidate(data)* bezeichnen wir im Folgenden den Prädikat, der uns angibt, ob die Daten für weitere Suche in Frage kommen. In unserem Beispiel von oben wären also *candidate("cb")=candidate("bc")=candidate("abdc")=true*.

Als logische Formel ergibt sich :

$$candidate(data) \leftrightarrow (bm(search) \rightarrow bm(data)) \tag{1}$$

Auflösung der Implikation:

$$candidate(data) \leftrightarrow (\overline{bm(search)} \vee bm(data)) \tag{2}$$

Doppelte Negation :

$$candidate(data) \leftrightarrow (\overline{\overline{\overline{bm(search)}} \vee bm(data)}) \tag{3}$$

De Morgan :

$$candidate(data) \leftrightarrow (\overline{bm(search) \wedge \overline{bm(data)}}) \tag{4}$$

Als Bitmuster gesehen :

$$candidate(data) \leftrightarrow ((bm(search) \wedge \overline{bm(data)}) = 0) \tag{5}$$

Dies führt uns zu der Überlegung den **negierten** Bitmuster (0=Pattern kommt vor und 1=Pattern kommt nicht vor) in der Datenbank abzuspeichern.
Die Formel 5 hat aus der Sicht der Implementierung noch weitere Vorteile gegenüber anderen Formeln:

- Moderne Prozessoren brauchen wenige Takte für AND Befehl
- Vergleich mit 0 ist schneller als Vergleich mit einer anderen Zahl.

Eine Implementierung in C++ könnte so aussehen.

```
...
 unsigned int position = 0;
 unsigned int bm_search = bm(search);
 while (bm_search & neg_bm_data(position))
 {
       position++;  // zum naechsten Datensatz gehen
 }
 ... hole den Datensatz und ueberpruefe ihn direkt.
```

3 Performance

Auf einer x86 CPU gibt es einen TEST Befehl (non destructive AND mit Flagsetzung), der genau die gewünschte Funktion durchführt. Das bedeutet, dass die Schleife aus 3 CPU-Befehlen aufgebaut werden kann(wenn Bitmuster aus max. 32 bit besteht). Obwohl die Suchzeiten linear steigen, ist die Suchzeit auch bei einer Million Datensätzen relativ gering. Durch optimierte Patterns kann man eine Trefferquote von über 90% erreichen (Anzahl der Kandidate dividiert durch die Anzahl der tatsächlichen Hits)